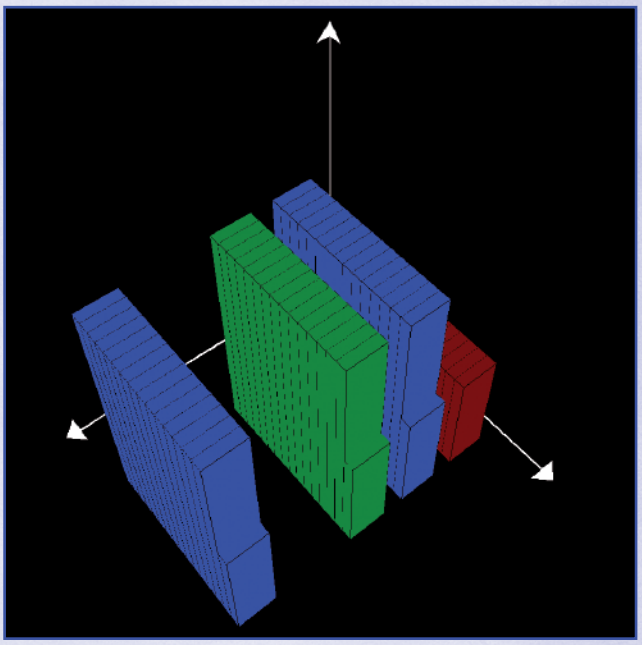
Rainer Buchty, Jan-Philipp Weiß (eds.)

High-performance and Hardware-aware Computing

Proceedings of the First International Workshop on New Frontiers in High-performance and Hardware-aware Computing (HipHaC'08)



universitätsverlag karlsruhe

Rainer Buchty, Jan-Philipp Weiß (eds.)

High-performance and Hardware-aware Computing

Proceedings of the First International Workshop on New Frontiers
in High-performance and Hardware-aware Computing (HipHaC'08)

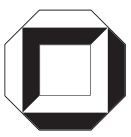
Lake Como, Italy, November 2008
(In Conjunction with MICRO-41)

High-performance and Hardware-aware Computing

Proceedings of the First International Workshop on New Frontiers in High-performance and Hardware-aware Computing (HipHaC'08)

Lake Como, Italy, November 2008
(In Conjunction with MICRO-41)

Rainer Buchty
Jan-Philipp Weiß
(eds.)



universitätsverlag karlsruhe

Impressum

Universitätsverlag Karlsruhe
c/o Universitätsbibliothek
Straße am Forum 2
D-76131 Karlsruhe
www.uvka.de

Universitätsverlag Karlsruhe 2008
Print on Demand

ISBN: 978-3-86644-298-6

Organization

Workshop Organizers:

Rainer Buchty
Karlsruhe Institute of Technology, Germany
Jan-Philipp Weiß
Karlsruhe Institute of Technology, Germany

Steering Committee:

Jürgen Becker
Karlsruhe Institute of Technology, Germany
Vincent Heuveline
Karlsruhe Institute of Technology, Germany
Wolfgang Karl
Karlsruhe Institute of Technology, Germany
Jan-Philipp Weiß
Karlsruhe Institute of Technology, Germany

Program Committee:

Mladen Berekovic
Universität Braunschweig, Germany
Alan Berenbaum
SMSC, USA
Nevin Heintze
Google Inc.
Vincent Heuveline
Karlsruhe Institute of Technology, Germany
Eric D'Hollander
Ghent University, Belgium
Ben Juurlink
TU Delft, The Netherlands
Wolfgang Karl
Karlsruhe Institute of Technology, Germany
Richard Kaufmann
Hewlett-Packard, USA
Paul Kelly
Imperial College, UK
Hsin-Ying Lin
Intel, USA
Rudolf Lohner
Karlsruhe Institute of Technology, Germany
Andy Nisbet
Manchester Metropolitan University, UK
Ulrich Rüde
Universität Erlangen-Nürnberg, Germany
Martin Schulz
LLNL, USA
Thomas Steinke
Zuse-Institut Berlin, Germany
Robert Strzodka
Max Planck Institut Informatik, Germany
Stephan Wong
TU Delft, The Netherlands

Preface

High-performance system architectures are increasingly exploiting heterogeneity: multi- and manycore-based systems are complemented by coprocessors, accelerators, and reconfigurable units providing huge computational power. However, applications of scientific interest (e.g. in high-performance computing and numerical simulation) are not yet ready to exploit the available high computing potential. Different programming models, non-adjusted interfaces, and bandwidth bottlenecks complicate holistic programming approaches for heterogeneous architectures. In modern microprocessors, hierarchical memory layouts and complex logics obscure predictability of memory transfers or performance estimations.

For efficient implementations and optimal results, underlying algorithms and mathematical solution methods have to be adapted carefully to architectural constraints like fine-grained parallelism and memory or bandwidth limitations that require additional communication and synchronization. Currently, a comprehensive knowledge of underlying hardware is therefore mandatory for application programmers. Hence, there is strong need for virtualization concepts that free programmers from hardware details, maintaining best performance and enable deployment in heterogeneous and reconfigurable environments.

The First International Workshop on New Frontiers in High-performance and Hardware-aware Computing (HipHaC'08) – held in conjunction with the 41st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-41) – aims at combining new aspects of parallel, heterogeneous, and reconfigurable system architectures with concepts of high-performance computing and, particularly, numerical solution methods. It brings together international researchers of all affected fields to discuss issues of high-performance computing on emerging hardware architectures, ranging from architecture work to programming and tools.

The workshop organizers would therefore like to thank the MICRO-41 Workshop Chair for giving us the chance to host this workshop in conjunction with one of the world's finest conferences on computer and system architecture – and of course all the people who made this workshop finally happen, most notably Wolfgang Karl (KIT) for initial inspiration. Thanks to the many contributors submitting exciting and novel work, HipHaC'08 will reflect a broad range of issues on architecture design, algorithm implementation, and application optimization.

Karlsruhe,
October 2008

Rainer Buchty & Jan-Philipp Weiß
Karlsruhe Institute of Technology (KIT)

Table of Contents

OROCHI: A Multiple Instruction Set SMT Processor

Takashi Nakada*, Yasuhiko Nakashima*, Hajime Shimada†, Kenji Kise‡ and Toshiaki Kitamura§
*Graduate School of Information Science, Nara Institute of Science and Technology, JAPAN
{nakada, nakashim}@is.naist.jp
†Graduate School of Informatics, Kyoto University, JAPAN
shimada@kuis.kyoto-u.ac.jp
‡Graduate School of Information Science and Engineering, Tokyo Institute of Technology, JAPAN
kise@cs.titech.ac.jp
§Graduate School of Information Sciences, Hiroshima City University, JAPAN
kitamura@arch.ce.hiroshima-cu.ac.jp

***Abstract*—To develop embedded computer systems, one straightforward way is to employ heterogeneous multi-processors or multi-cores that have a single traditional core and several SIMD/VLIW cores. This approach is suitable not only for quick integration of de-facto OS and new multimedia programs, but also for QoS. However, such well-known architecture increases the area, the complexity of the bus structure, the cost of the chip and the inefficient use of the dedicated cache memory. As an efficient embedded processor, we propose a heterogeneous SMT processor that has two different front-end pipelines. Each pipeline corresponds to ARM architecture for irregular programs and FR-V (VLIW) architecture for multimedia applications. FR-V instructions run through the simple decoder and are enqueued into the VLIW queue. The instructions in the VLIW queue are simultaneously shifted to the next stage after the instructions at the final portion are all issued. On the other hand, ARM instructions are decomposed into simple instructions suitable for the VLIW queue. The instructions are scheduled based on the data dependencies and the empty slots. After that, the mixed instructions in the VLIW queue are issued to the common back-end pipeline. In this paper, a simple instruction scheduler and a mechanism for QoS are presented. We evaluated the performance with an RTL-level simulator and evaluated the chip area. The results show that the microarchitecture can increase the total IPC by 20.7% compared to a well-known QoS mechanism controlled by a process scheduler in OS, and can reduce the total chip area by 34.5% compared to a well-known separated multi-core implementation.**

***Index Terms*—Heterogeneous SMT processor, VLIW, Quality of Service**

I. Introduction

In recent years, it has become popular to enjoy high-quality multimedia contents via portable devices. The processors for such embedded devices are required to accomplish high performance for multimedia applications and work on ultra low-power to enable use of smaller batteries. Unfortunately, well-known superscalar processors are unacceptable for such embedded devices on two counts. First, power-hungry processors with large heatsinks are hard to fit into the embedded devices that are usually composed in a small chassis. Second, the processors need to consume less power so as to extend battery life as much as possible. For this field, in place of traditional wide issue superscalars, many heterogeneous multi-cores have been proposed to meet the requirements. Considering the heavy multimedia workload in modern embedded devices, VLIW processors are good candidates because enough ILP in multimedia programs is easily detected by the compiler so that complicated issue mechanisms can be omitted. By incorporating well designed multimedia libraries, VLIW can achieve good performance with low power. However, VLIW is less competitive in applications with few ILP. Moreover, library support for general purpose applications is comparatively poor. Consequently, some general purpose processors are also included. This results in a heterogeneous multi-core processor. Thus, heterogeneous multi-core processors have become popular, as is the case with the Cell Broadband Engine [1], which includes a POWER Processing Element (PPE) as a general processor and eight Synergistic Processing Elements (SPEs) as media processors.

However, from the point of view of semiconductor technology, multi-cores that increase the footprint by incorporating discrete cores straightforwardly are not the best solution because static power leakage and process variation will be big obstacles in next generation low power and high-performance processors. Static power leakage is in proportion to footprint. It is crucial to reduce the footprint in the near future. In particular, the general processor in the multi-core is quite large because its design tends to be imported from traditional implementations, despite the small performance contribution of the general processor. If we unify the general purpose processor with media processors such as VLIW, the footprint is minimized, and the dedicated cache area of the general purpose processor can be effectively utilized as an additional cache for the media processors. Such integration shows promise in the field of smaller footprints and high-performance. Meanwhile, conventional SMT execution models [2], which also share a single pipeline and the data cache, are not suitable for QoS control in general. However, in many embedded systems, QoS control is one of the important requirements. The processor has to guarantee the frame rate for a video decoder, for example. The heterogeneous SMT for embedded processors should meet

demands such as these that are not popular in conventional SMT.

Therefore, we propose a heterogeneous SMT processor named OROCHI, which can execute simultaneously both the conventional instruction set and the VLIW instruction set. By unification of the back-end pipeline, which includes a load/store unit, the processors based on different architecture share execution units and a data cache. Each processor has the opportunity to use more cache area during the time that the other processor does not need a large cache area. First, we propose a novel QoS-aware instruction scheduling mechanism with a VLIW queue that is completely different from traditional superscalar processors. It schedules VLIW instructions directly and also transforms conventional instructions efficiently. Conventional instructions are decomposed into simple instructions and inserted into the empty slot of the VLIW queue. Second, we adopt a cache miss prediction mechanism incorporated in branch predictors and a selective instruction flush mechanism in the VLIW queue, which are made more effective than previous QoS control mechanisms by using an OS scheduler [3] or some hardware approach such as dynamic cache partitioning [4].

The rest of this paper is organized as follows. Section 2 gives an overview of OROCHI. Section 3 reveals the microarchitecture of OROCHI. Section 4 describes its evaluation. Finally, Section 5 concludes the paper and describes future work.

II. Previous Work on QoS (Quality of Service)

To sustain the QoS, several methods are proposed. These approaches are classified into two categories, a software approach and a hardware approach.

The most traditional and common software approach is scheduling by an OS. However, reducing the execution time of other applications is the only way to improve the performance of the QoS-aware applications. With monitoring performance counters, IPC, etc., an OS can sustain the fairness to some extent [3]. However, the performance of each application tends to be degraded. Therefore, it is hard to sustain the QoS by the scheduler.

Hardware approaches are more powerful than OS approaches, one being a cache partitioning [5] that divides the cache memory to achieve a dedicated cache for each application. The dedicated cache is effective in alleviating the interaction among applications. However, each cache size thus decreases to less than the total cache size, as a result of which, the performance is to an unacceptable degree decreased [6]. To alleviate this problem, dynamic cache partitioning [4], which adjusts the boundaries of a cache, and virtual private caches [7], which control cache bandwidth, have been proposed. Unfortunately, their effectiveness is also limited.

A central problem in QoS resides in pipeline stalls due to unexpected cache misses. So, some cache miss prediction mechanism shows promise for sustaining QoS. For instance, Compaq Alpha 21264 [8] has a cache hit/miss predictor for a speculative issue mechanism. If the cache is predicted as hit, to minimize load-use latency, Alpha 21264 issues instructions speculatively that depend on the previous load instruction. If the speculation fails, all integer pipelines are rewound. The pipelines are restarted and the instructions are reissued. To reduce this overhead, the cache hit/miss predictor is very important for Alpha 21264.

Another approach is a selective instruction flush mechanism [9]. When a cache miss occurs on some thread, instructions that depend on the load instruction are removed from an instruction window to avoid unnecessary resource occupation. After the cache is filled, the removed instructions are refilled to the instruction window.

III. Microarchitecture of OROCHI

Recent embedded devices that deal with high-quality multimedia contents have a conventional processor (scalar processor) and a media processor (ex. VLIW). The conventional processor usually executes OS codes and miscellaneous low ILP applications. To minimize developing time, exploiting conventional processors is crucial, so many legacy codes and libraries are required to complete the system. On the other hand, some media processor is required to accelerate the media processing. There is much data parallelism in multimedia applications, so typical media processors employ an effective instruction set such as VLIW, SIMD, etc. that can easily exploit data parallelism at a low hardware cost. We considered that the legacy codes can be transformed to fit to some VLIW structure to reduce the footprint of the total system. We evaluated a heterogeneous SMT comprising ARM [10] architecture, as one of the most popular embedded processors with de-facto OS, and FR-V [11] architecture, as another popular embedded processor for the image processing field.

FR550 is an eight issue FR-V architecture processor. FR550 can issue four integer instructions and four floating point instructions or media instructions simultaneously. The media instructions support saturation operation, multiply and accumulate, SIMD, and so on. Branch and load/store instructions are classified as integer instruction. It can issue two branch instructions simultaneously to support a three-way branch and also two load/store instructions.

Figure 1 outlines the concept of OROCHI with a VLIW back-end pipeline based on FR550. The most important difference with a popular pipeline is a **VLIW queue** that holds two different instruction sets simultaneously. The key point of this structure is that some empty slots always exist in the queue. Because the number of function units of a VLIW processor is usually more than the maximum number of instructions in one VLIW. As a result, even if the VLIW instruction stream executes high performance multimedia applications that occupy almost all of the instruction slots, enough empty slots remain for execution legacy codes of ARM applications or OS. Therefore, we considered that it is possible to integrate the two different types of processors effectively without performance degradation.

In detail, the back-end pipeline is comprised of instruction queue, register file, execution units, data cache and I/O inter-

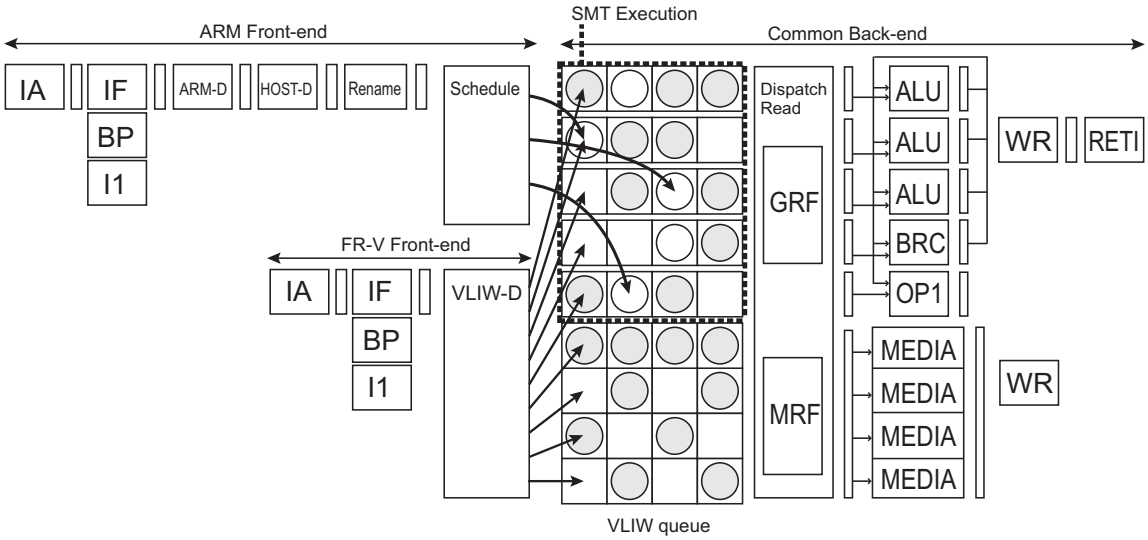


Fig. 1. Pipeline of OROCHI

face. A different type of instruction set is translated to fit the back-end pipeline. Several front-end pipelines are connected to the instruction queue. Thus, some kinds of processors can be united with small cost.

A. Outline of the pipeline

OROCHI has two front-end pipelines. Each front-end has an instruction address generator (IA), an instruction fetch stage (IF) with a cache (I1) and a branch predictor (BP) that includes a load-use miss predictor described later, a decoder (ARM-D, HOST-D corresponding to instruction decomposition similar to Intel P6 architecture [12] or Netburst architecture [13], and VLIW-D). Additionally, ARM front-end has a rename stage (Rename) for out-of-order execution. The decoded instructions from VLIW-D are directly enqueued into the left-most portion of the queue. Meanwhile, the renamed instructions from Rename are scheduled to the queue based on the data dependencies and the empty slots (Schedule). The detailed mechanism of such scheduling is described later.

The instructions, which have architecture flags to distinguish between ARM and FR-V instructions, in the queue are shifted toward the execution units simultaneously when the instructions in the right-most columns are all issued. The issue mechanism is very similar to the popular VLIW architectures. Obviously, it is very important to schedule instructions so that interlocks in the instruction queue seldom occur because partial data dependency interlocks the whole of the queue.

The back-end pipeline is based on VLIW, as mentioned, and includes three integer units with shifter and partial multiplication functions (ALU), one load/store unit (OP1), one branch unit (BRC) and four media units dedicated to FR-V instruction streams (MEDIA)[1]. These function units are a subset of the FR550 processor. All function units except MEDIA units are shared by ARM and FR-V. The back-end pipeline also has a general register file (GRF), which has eight read ports and five write ports, and a media register (MRF), which has eight read ports and four write ports. Since renaming is not necessary for in-order execution of FR-V, only a logical register file is required for FR-V. Even though logical register spaces are separated between ARM and FR-V, the register file is shared so that the size of the register file becomes large. However, numbers of read and write ports are not increased. Since OROCHI does not have a register transfer instruction between the general register and media register, media register file is independent from general register file.

As for ARM instructions, the results are written in the reorder buffer out-of-order (WR) and then completed in-order in the following retire stage (RETI). As for FR-V instructions, the results are written in the architecture registers and also completed in-order.

When branch prediction misses in a thread, the related instructions are flushed from the front-end and the instruction queue, while the other thread keeps executing the instruction stream.

B. QoS Aware Instruction Scheduling

In the conventional SMT, the requirements for QoS are not so strong because the fairness between processes is the most important issue for the system. Besides, in an embedded system area, special considerations are required to maintain QoS for certain multimedia applications.

Under typical usage of OROCHI, the processor executes both the multimedia processing thread written in the VLIW instruction set and the OS thread written in the conventional instruction set simultaneously. From the multimedia processing side, there are many deadlines. The processor has to guarantee the completion of the task before the deadline to meet the media QoS requirement.

[1]Floating point units are not included. ARM and FR-V use a soft-float library instead.

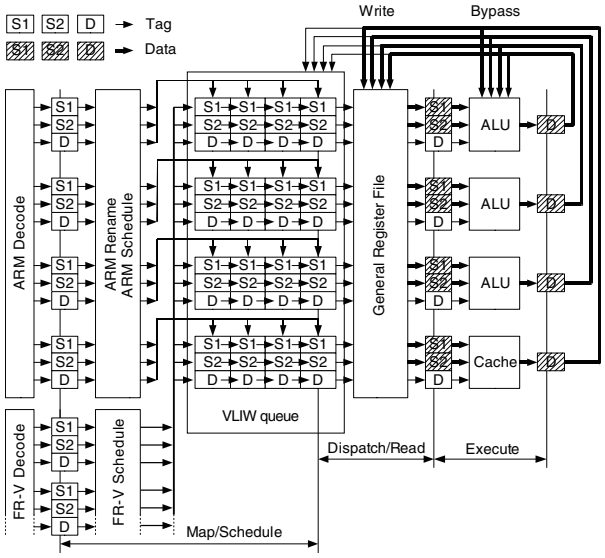


Fig. 2. Detailed Structure of OROCHI

The excessive method to maintain QoS of multimedia applications running on FR-V is complete shutdown of ARM instruction streams. However, this is not acceptable from the point of view of real-time interruption handling.

It is known that the compiler for VLIW schedules instructions statically. If the compiler cannot find an instruction to fill slots, empty slots are left as NOPs because the VLIW does not schedule the instructions dynamically. Even if the ARM instructions are inserted into such empty slots, performance drop never occurs on the condition that ARM instructions do not interfere with the instruction streams of FR-V applications. The most reasonable technique to sustain the performance of FR-V is to provide enough slots for FR-V and to schedule ARM into the unused slots.

Figure 2 describes the structure of instruction scheduling. In this figure, we omit media and branch units to simplify. At first, if the left-most portion is empty, FR-V instructions are enqueued into the left-most portion of the queue. Then ARM instructions are inserted into the queue. To find a suitable empty slot, the scheduler compares the destination register numbers of instructions that are already scheduled in the queue and source register numbers of the instruction to be scheduled, then inserts into a suitable slot nearest to the corresponding execution unit as possible. The scheduling mechanism allows out-of-order execution with the preceding rename stage and achieves comparable performance with out-of-order superscalar processors. After that, these mixed instructions in the queue are issued to the common back-end unit in the same manner as VLIW.

C. Issue Instructions

In the dispatch stage, VLIW hardware dispatches all of the instructions in the right-most portion of the queue. If there is an instruction that cannot be issued due to unresolved data dependency, such as load-use that has a possibility of cache miss, the following pipeline stages stall. When an L1 data cache miss occurs, it stalls not only the dependent instructions but also instructions in the same line. Such simple structure results in lower complexity than superscalars that incorporate complicated wakeup and select logics. Instead, performance seriously drops when one of the instructions waits for the data produced by previously dispatched instructions. The major event of such a stall derives from a data cache miss. In the traditional instruction scheduling, in order to greatly reduce the execution latency, the instructions that require some load data are scheduled as if there were no cache miss reported. Conversely, OROCHI should insert ARM instructions without interference to FR-V. We only have a limited instruction scope, so that there is a high probability of pipeline hazards due to L1 data cache misses. Basically, OROCHI maintains QoS of the FR-V application by scheduling ARM instructions carefully. The key ideas of the mechanism are cache miss prediction and selective instruction flush described in next section.

D. QoS Control with Cache Miss Prediction and Selective Instruction Flush

To alleviate this pipeline stall problem, we propose a cache hit/miss predictor and a selective instruction flush mechanism.

In general, the cache miss predictor indicates whether the target cache access will hit or miss. However, OROCHI has to control not only where the depended instruction should be scheduled in the queue but also when it should be scheduled. For instance, when a cache miss is predicted, instructions that depend on the load data should be scheduled apart from the load instruction. If we cannot find a suitable free slot in the instruction window because a long delay is predicted, the instruction should be delayed to schedule. Such a mechanism has the potential to avoid pipeline stall due to the cache misses, if it can learn cache behavior efficiently.

Conversely, if the prediction is incorrect, the processor cannot avoid a pipeline stall. To alleviate this case, we propose an additional selective instruction flush mechanism. When an ARM load instruction results in a cache miss, all ARM instructions that include the load instruction are purged from the instruction window. Note that the cache fill request is not canceled. Since all instructions have an architecture flag, it is easy to find the ARM instructions. After that, the load instruction and the following are scheduled again. With the mechanism, the pipeline stall is eliminated and the FR-V instructions are executed without interference from ARM.

Figure 3 outlines these techniques. In this figure, the **LUMP** indicates a Load-Use Miss Predictor, which predicts whether an instruction will bring cache miss or not. The LUMP is implemented in the branch predictor. Note that at this stage, it is unknown whether the instruction is a load or not. Instead, the hardware cost is minimized by sharing the table with the branch predictor. The additional information to the PHT of the Gshare branch predictor is several bits that indicate the estimated delay cycles to schedule. When a load instruction is scheduled, the scheduler controls the insert point and the timing according to the prediction (1a). For example, when

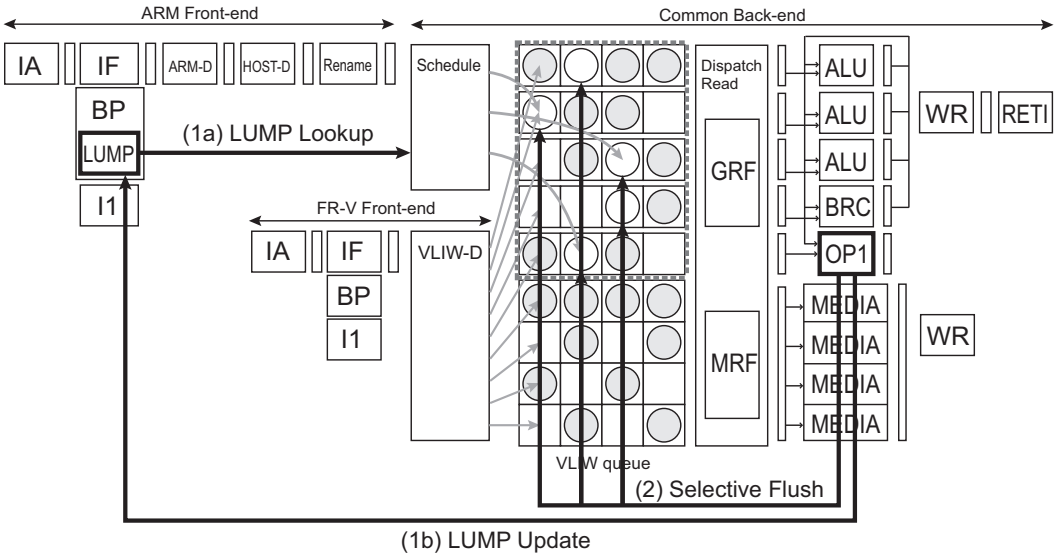


Fig. 3. LUMP & Selective Flush

the corresponding counter indicates 3, three cycles of cache delay can be hidden, but if the cache access is hit, the three cycles become a penalty. When a load instruction is executed, the corresponding entry of the table is updated (1b). In detail, when a load instruction leads to cache miss, the corresponding counter is incremented and vice versa. If the selective instruction flush mechanism is enabled also, the cache miss leads to flushing all of the ARM instructions including the load instruction from the VLIW queue (2).

IV. Evaluation

We evaluate the multiple instruction set SMT processor OROCHI from the view of IPC and the feasibility. First, the performance of the VLIW queue is evaluated as compared with an out-of-order superscalar processor. Second, SMT performance with both ARM and FR-V applications is evaluated. Finally, the QoS features are measured. Table I shows the basic parameters of OROCHI.

A. VLIW Queue

We preliminarily evaluate the performance of the VLIW queue as compared with a superscalar processor using an RTL-level simulator. We also design with ASIC (0.25μm technology) to evaluate the delay and the area.

Also for the evaluation of the VLIW queue, we design another ARM superscalar processor with a centralized instruction window as a baseline (ARMSS). Figure 4 outlines the baseline processor. The fetch, decode, decompose and back-end units are the same as OROCHI's. However, ARMSS has a centralized instruction window in order to support dynamic out-of-order execution. ARMSS also has complicated **Wakeup-Select** logic. The Wakeup-Select logic searches for instructions that are ready to be issued (Wakeup) and decides which instructions are issued from the candidates (Select) within one cycle. In Figure 4, we can find an additional large selector in the Select/Read stage. We compare IPC using several programs from MiBench [14] running on ARM.

TABLE I
Evaluation parameters

Cache miss predictor	PHT: additional 3bit × 8K entries (integrated in the branch predictor)
Branch predictor	PHT: 2bit × 8K entries (gshare)
Return Address Stack	8 entries
Physical register	32 entries
Store buffer	8 entries
Cache line size	64byte
ARM I1 cache	4way, 16KB miss latency 8cycle
FR-V I1 cache	4way, 16KB miss latency 8cycle
D1 cache	4way, 32KB miss latency 8cycle
Unified L2 cache	4way, 2MB miss latency 40cycle
VLIW queue	depth 4

TABLE II
Performance of ARM Superscalar (ARMSS)

	IPC	delay [ns]	Freq [MHz]	IPC×Freq [MIPS]
ARMSS	1.335	13.51	74.0	98.8(1.00)
OROCHI	1.331	8.54	117.1	155.9(1.58)

Table II shows the IPCs, the circuit delays and the overall performances. Table III shows the areas. From these results, OROCHI outperforms ARMSS. The comparison of IPCs shows that ARMSS gains only 0.3% over OROCHI. The comparison in the delay shows that OROCHI is faster than ARMSS by 36.8% due to the simple instruction issue mechanism. As a result, the overall performance of OROCHI is expressed as the product of IPC and frequency is superior

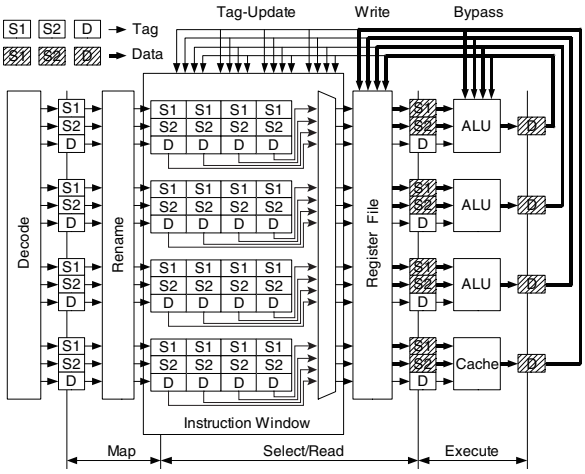


Fig. 4. Pipeline of ARM Superscalar (ARMSS)

TABLE III
AREA OF ARM SUPERSCALAR (ARMSS)

	Relative cell area
ARMSS	1.000
OROCHI (ARM only)	1.016

to ARMSS by 57.8%. The comparison in the area shows that sizes of two implementations are almost the same. After that, OROCHI is found to be an efficient implementation as an out-of-order design.

B. Focus on IPC

The overall performance of the SMT is evaluated using an RTL-level simulator that has a capability to run the real μClinux/ARM [15] with no MMU. Some benchmarks from MiBench are compiled as an ARM binary or an FR-V binary respectively and run simultaneously with 'small' datasets under the control of the OS. We select some irregular applications (e.g. bitcount and dijkstra) for ARM and 13 media applications for FR-V. The average IPC is measured from the point from which both programs start to the point at which the FR-V program terminates while the ARM program is executed repeatedly.

Figures 5 and 6 show the average IPCs, which includes a baseline comprised of the total IPC of separated execution of ARM and FR-V where some heterogeneous multi-core configuration is assumed. The leftmost bars of each result show the baseline (oracle) IPCs that correspond to simple summation of the IPCs of ARM and FR-V. The rest of the bars of each show IPCs of SMT execution. Note that the IPC of ARM includes the execution of OS codes.

With ARM bitcount, the IPC of FR-V achieves 98.3% of the ideal performance and the IPC of ARM results in 73.4%. In the same manner, with ARM dijkstra, the IPC of FR-V achieves 87.4% of the ideal and the IPC of ARM results in 76.4%. These results clearly show that OROCHI can successfully unite the two different types of processors in a single pipeline.

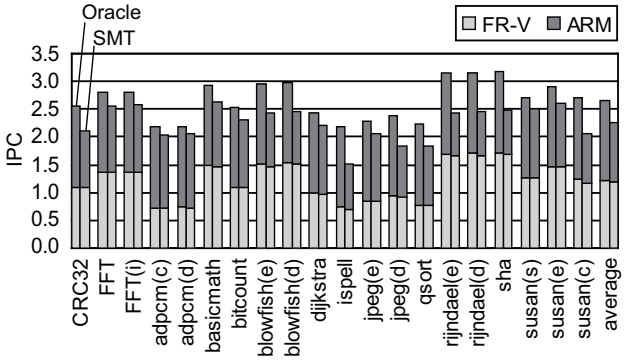


Fig. 5. ARM bitcount & FR-V MiBench

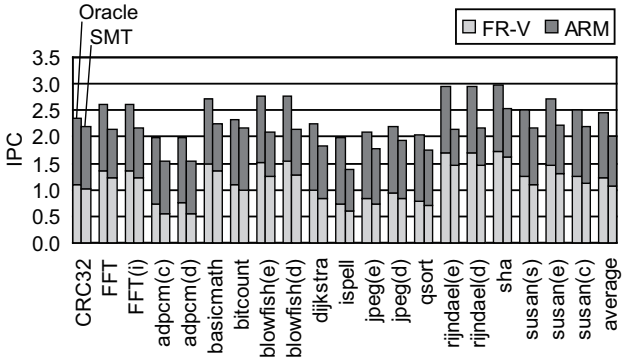


Fig. 6. ARM dijkstra & FRV-MiBench

Considering the difference of the performance between ARM bitcount and ARM dijkstra, some difference of data cache miss ratio is observed. Although the miss ratio of the first level data cache in bitcount is only 0.7%, the miss ratio in dijkstra is 5.5%. The difference in memory pressure is considered to be a major reason for the phenomenon.

C. Focus on QoS

In contrast to the assumption in the previous section, it is easily imagined that the ARM programs with high memory pressure interfere with the performance of FR-V. To alleviate this case, we propose two key hardware mechanisms we call 'cache miss predictor' and 'selective flush' as mentioned before, which are more effective than the software approaches.

For the comparison, an OS-based QoS mechanism inspired by a previous work [3] is evaluated. In this mechanism, the process scheduler in OS controls the priority of ARM programs so that the FR-V programs maintain the performance to some extent.

Figure 7 shows the results. Oracle, Base, LUMP, Flush, LUMP+Flush and OS Sched. correspond to oracle performances, without any additional mechanism, using load-use miss prediction (LUMP), selective flush, both the LUMP and the Flush, and the OS scheduler respectively. TOTAL IPC shows the sum of the ARM IPC and the FR-V IPC. The Oracle and the Base are the same as the results in Figure 6.

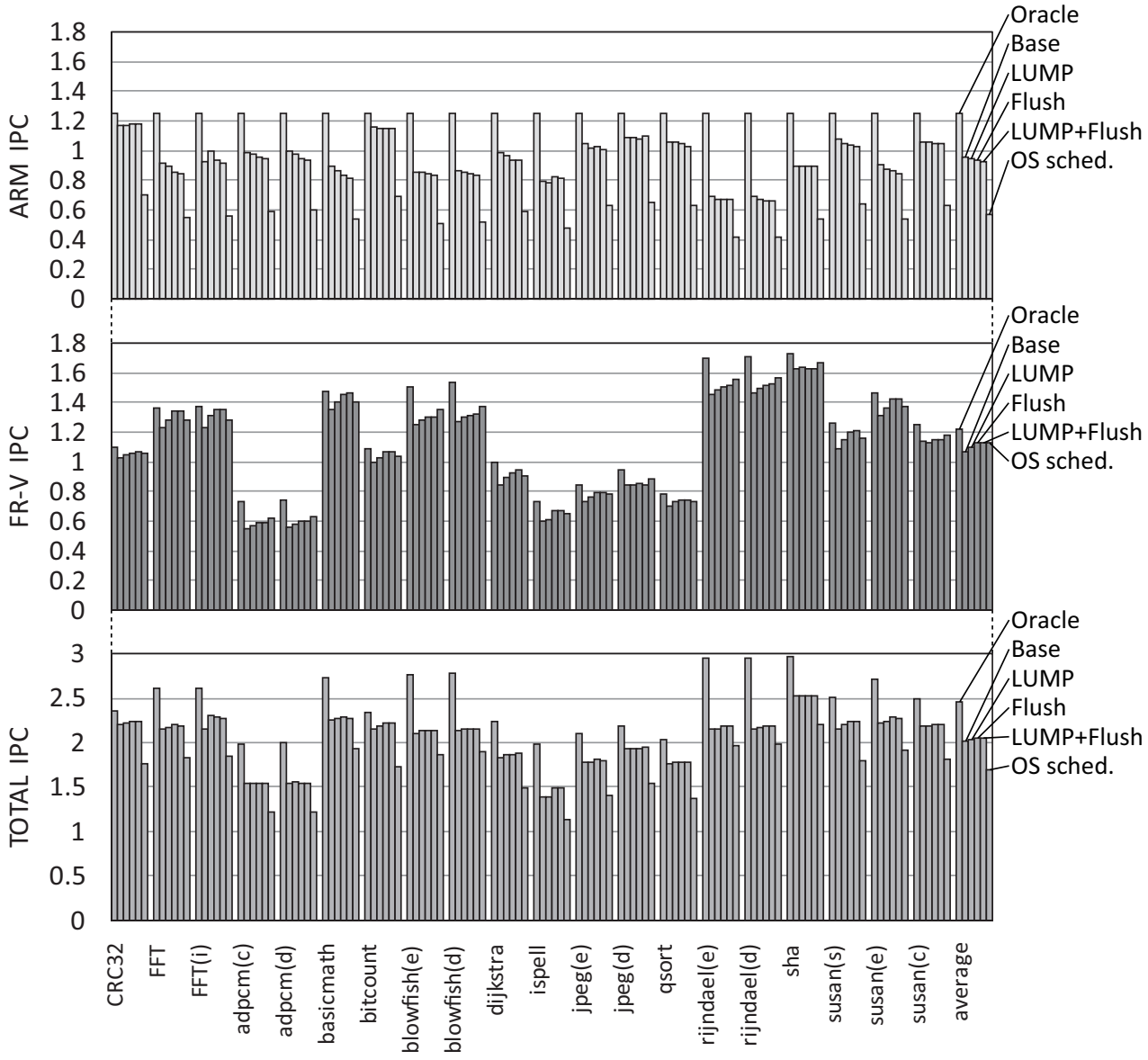


Fig. 7. QoS Assurance of FR-V (w/ ARM dijkstra)

With LUMP or Flush, the performance of FR-V (FR-V IPC)is increased from 87.4% (Base) to 90.1% and 92.5% on average respectively. Moreover, when both the LUMP and the Flush techniques are applied (LUMP+Flush), it achieved 92.8% on average, whereas the amount of the decrease in ARM performance (ARM IPC) corresponds to the amount of the increase in FR-V performance. Consequently, the total performance (TOTAL IPC) does not decrease at all. In addition, using the OS scheduler (OS Sched.), the performance of FR-V reached 92.5% on average. However, in order to achieve this performance, the OS scheduler limits ARM execution time by 60.0%. Therefore the performance of ARM is significantly decreased by 60.0% of Base and consequently the total performance is only 82.9% as compared with our hardware mechanism.

After that, the result shows LUMP and the selective flush mechanism are efficient for sustaining QoS. In particular, the latter can increase IPC of FR-V by 5.1%. Note that the total performance is not decreased.

D. Feasibility Study

To evaluate the effectiveness of unification quantitatively, we designed OROCHI using ASIC (0.25μm rule). Table IV shows the comparison of several types of cores. OROCHI indicates the entire area of OROCHI, OROCHI (FRV only) and OROCHI (ARM only) indicate OROCHI without ARM front-end and FRV front-end respectively. The differences of these results correspond to the size of the ARM front-end

TABLE IV
AREA OF OROCHI

Configuration	Relative cell area
OROCHI	1.000
OROCHI (FRV only)	0.668
OROCHI (ARM only)	0.859
ARM front-end	0.332
FRV front-end	0.141
Common back-end	0.527

(33.2%) and FR-V front-end (14.1%). The ARM front-end is twice as big as the FRV front-end due to renaming and out-of-order execution mechanisms. However, note that the difference of the area is emphasized because of the small cache (L2 is not included) and lack of floating point units as mentioned. If we make a heterogeneous multicore using this front-end and back-end, the size must be 152.7% due to redundant back-end; thus, OROCHI can reduce the chip area by 34.5%. Assuming the same semiconductor technology, OROCHI is comparable to only one SPE of a Cell Broadband Engine in size.

V. CONCLUSIONS AND FUTURE WORK

In this paper, we proposed a heterogeneous SMT processor OROCHI that can execute both a conventional instruction set and a VLIW instruction set simultaneously.

By unification of the back-end pipeline that includes a load/store unit, the processors based on different architecture can share execution units and a data cache. Each processor has an opportunity to use more cache area while the other processor does not need a large cache area. First, we proposed a novel QoS aware instruction scheduling mechanism with a VLIW queue. It schedules VLIW instructions directly and also transforms conventional instructions efficiently. The latter instructions are decomposed and inserted into the empty slot of the VLIW queue. Second, we adopted a cache miss prediction mechanism and a selective instruction flush mechanism in a VLIW queue that are more effective than OS-based QoS.

We evaluated the performance with an RTL-level simulator with MiBench and OS. The result shows that the microarchitecture can achieve 98.3% of the ideal FRV performance and 73.4% of the ideal ARM performance simultaneously when executing a light ARM process. Even if it executes a heavy ARM process, the QoS is maintained by 92.8% of FRV performance. As compared to a well-known QoS mechanism controlled by a process scheduler in OS, this microarchitecture can increase the total IPC by 20.7%. We also evaluated the chip area by designing the microarchitecture on ASIC. The result shows that it can successfully share back-end, which accounts for 52.7% of the chip area. As a result, the microarchitecture can reduce the total chip area by 34.5% compared to well-known separated multi-core implementation.

As future work, we will measure the real power consumption of OROCHI to evaluate the reduction of the power consumption, which includes static power leakage.

ACKNOWLEDGMENT

This research is joint research with Semiconductor Technology Academic Research Center and partially supported by the Ministry of Education, Science, Sports and Culture, Grant-in-Aid for Scientific Research (B), 19300012, 2006.

REFERENCES

[1] D. Pham *et al.*, "The design and implementation of a first generation cell processor," in *ISSCC*, 2005, pp. 184–592.
[2] J. A. Brown and D. M. Tullsen, "The shared-thread multiprocessor." in *ICS*, 2008, pp. 73–82.
[3] A. Fedorova, M. Seltzer, and M. D. Smith, "Improving performance isolation on chip multiprocessors via an operating system scheduler," in *PACT*, 2007, pp. 25–38.
[4] J. Chang and G. S. Sohi, "Cooperative cache partitioning for chip multiprocessors," in *ICS*, 2007, pp. 242–252.
[5] S. E. Raasch and S. K. Reinhardt, "The impact of resource partitioning on smt processors," in *PACT*, 2003, pp. 15–25.
[6] R. R. Iyer *et al.*, "Qos policies and architecture for cache/memory in cmp platforms," in *SIGMETRICS*, 2007, pp. 25–36.
[7] K. J. Nesbit, J. Laudon, and J. E. Smith, "Virtual private caches," in *ISCA*, 2007, pp. 57–68.
[8] R. E. Kessler, "The alpha 21264 microprocessor," *IEEE Micro*, vol. 19, no. 2, pp. 24–36, 1999.
[9] A. R. Lebeck *et al.*, "A large, fast instruction window for tolerating cache misses," in *ISCA*, 2002, pp. 59–70.
[10] *ARM Architecture Reference Manual, ARM DDI0100E*, ARM Limited, 2000.
[11] *FR550 Series Instruction Set Manual Ver.1.1*, FUJITSU Limited., 2002.
[12] L. Gwennap, "Intel's p6 uses decoupled superscalar design," *Microprocessor Report*, vol. 9, no. 2, pp. 9–15, 1995.
[13] G. Hinton *et al.*, "The microarchitecture of the pentium4 processor," in *Intel Technology Journal, Q1*, 2001.
[14] M. Guthaus *et al.*, "Mibench: A free, commercially representative embedded benchmark suite," in *IEEE 4th annual International Workshop on Workload Characterization*, 2001.
[15] "μclinux," http://www.uclinux.org/.

Experiences with Numerical Codes on the Cell Broadband Engine Architecture

Markus Stürmer, Daniel Ritter, Harald Köstler, and Ulrich Rüde
System Simulation Group
Department of Computer Science
University Erlangen-Nuremberg
Cauerstraße 6, 91058 Erlangen
markus.stuermer@informatik.uni-erlangen.de

Abstract

Many numerical computations in science and engineering require high memory bandwidth and computational power. The Cell Broadband Engine Architecture (CBEA), a heterogeneous multicore architecture, promises both. We evaluated the potential of the CBEA for numerical codes in the areas image processing, computational fluid dynamics, and molecular dynamics. We present results and derive the strengths and challenges for using this novel architecture.

Keywords: CBEA, Cell processor, performance optimization, image processing, computational fluid dynamics, molecular dynamics

1. Introduction

Multicore architectures are the current trend to serve the insatiable demand for computational power in science, engineering, economy, and gaming. In contrast to other chip designers that put multiple, basically identical cores on a chip, STI[1] took a different approach with their Cell Broadband Engine Architecture (CBEA) that promises outstanding performance by establishing a heterogeneous design, whose key concepts are outlined in Sect. 2. The first machine to break the Petaflop barrier in Linpack was built of 12,960 PowerXCell 8i, the latest implementation of the CBEA, and 6,480 AMD Opteron processors at the Los Alamos National Laboratory.

To explore the potential of this novel architecture for numerical applications, we describe performance-optimized implementations on the CBEA for applications in image processing (Sect. 3), computational fluid dynamics (Sect. 4), and molecular dynamics (Sect. 5) before recapitulating the special features of the architecture in Sect. 6.

[1] Sony, Toshiba and IBM

2. Architectural overview

The first implementation of the CBEA, the so-called Cell Broadband Engine (Cell/BE) is used e. g. in the Sony Playstation™ 3 game console and IBMs QS20 and QS21 blades. Its organization is depicted in Fig. 1 [5, 6]: The backbone of the chip is a fast ring bus—the Element Interconnect Bus (EIB)—connecting all units on the chip and providing a throughput of up to 204.8 GB/s in total when running at 3.2 GHz. A PowerPC-based general purpose core—the Power Processor Element (PPE)—is primarily used to run the operating system and control execution, but has only moderate performance compared with other general purpose cores. The Memory Interface Controller (MIC) can deliver data with up to 25.6 GB/s from Rambus XDR memory and the Broadband Engine Interface (BEI) provides fast access to I/O devices or a coherent connection to other Cell processors. The computational power resides in eight Synergistic Processor Elements (SPEs), simple but very powerful co-processors consisting of three components: Synergistic Execution Unit (SXU), Local Storage (LS), and Memory Flow Controller (MFC).

The SXU is a custom Single Instruction Multiple Data (SIMD) only vector engine with a set of 128 128-bit-wide registers and two pipelines. It operates on 256 kB of its own LS, a very fast, low-latency memory. SXU and LS constitute the Synergistic Processor Unit (SPU), which has a dedicated interface unit connecting it to the outside world: the primary use of the MFC is to asynchronously copy data between LS and main memory or the LS of other SPEs using Direct Memory Access (DMA). It also provides communication channels to the PPE or other SPEs and is utilized by the PPE to control execution of the associated SPU. Each

SPE can be seen as a very simple computer performing its own program, but dependent on and controlled by the PPE.

The Cell/BE is able to perform 204.8 GFlop/s using fused-multiply-adds in single precision (not counting the abilities of the PPE), but is limited regarding double precision. Only six SPEs are available under Linux running as a guest system on the Sony Playstation™ 3, what reduces the maximum performance there accordingly to 153.6 GFlop/s. The newer PowerXCell 8i [7], used in IBMs QS22 blades, differs from the older Cell/BE by SPEs with higher performance in double precision (12.8 instead of 1.8 GFlop/s each) and a converter that allows connecting DDR2 memory to the MIC.

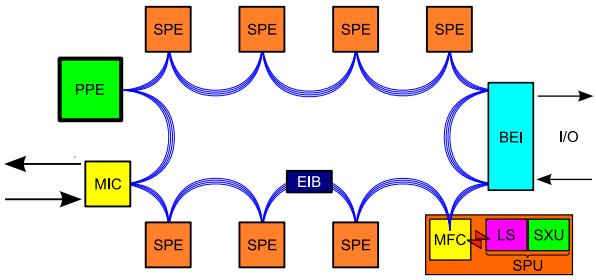


Figure 1. Schematic view of the STI Cell Broadband Engine.

While standard PowerPC software and compilers can be executed on the PPE's computation unit (the PowerPC Processor Unit, PPU), software must be adapted to take advantage of the SPEs, whose SXUs use their own instruction set. The basic approach to write CBEA-enhanced software is to separately implement the parts running on PPE and the SPEs, where libraries and language extensions help in issuing and waiting for DMA transfers, and doing the communication and synchronization between the different agents. From a software perspective, a program running on the PPU acquires an SPE and loads a code image to its LS first. To actually start the program on the SPE, a system call is used, which does not return until the SPE code suspends execution or terminates.

There are several general or Cell-specific approaches to ease the creation of heterogeneous parallel software, like IBM's Accelerated Library Framework (ALF) and Data Communication and Synchronization (DaCS) library, Cell Superscalar (CellSs) by the Barcelona Supercomputing Center, the RapidMind Multi-Core Development Platform, or Mercury's Multicore Plus SDK, only to mention some of them.

3. Image processing

Image processing is one of the applications for which the Cell/BE is especially suitable. Images have naturally regular data structures and are processed using regular memory accesses so that data can be transferred easily by DMA. Additionally, single precision is usually sufficient for image processing tasks. Besides the traditional techniques for image and video compression based e. g. on wavelets and Fourier transforms, methods using partial differential equations (PDEs) have been developed. These methods have the potential for providing high quality, however they are usually very compute intensive.

The PDE-based video codec PDEVC [10] is conceptionally very simple. For each picture, typically 10–15% of the pixels of an image are selected and stored. All remaing pixels are discarded and must therefore be reconstructed in the decoding stage. We will not discuss the algorithms for selecting the so-called *landmark pixels*, but will rather focus on the core algorithm used in the reconstruction phase, when the landmarks Ω_c and the corresponding pixel values u^0 are given. Filling in the missing pixels is the so-called *inpainting problem* [3], which is modeled by a partial differential equation of the form

$$\begin{aligned} -\operatorname{div}\left(D_{u_\sigma}\nabla u\right) &= 0 \quad \text{if } \mathbf{x} \in \Omega \setminus \Omega_c \\ u &= u^0 \quad \text{if } \mathbf{x} \in \Omega_c \end{aligned} \quad ,$$

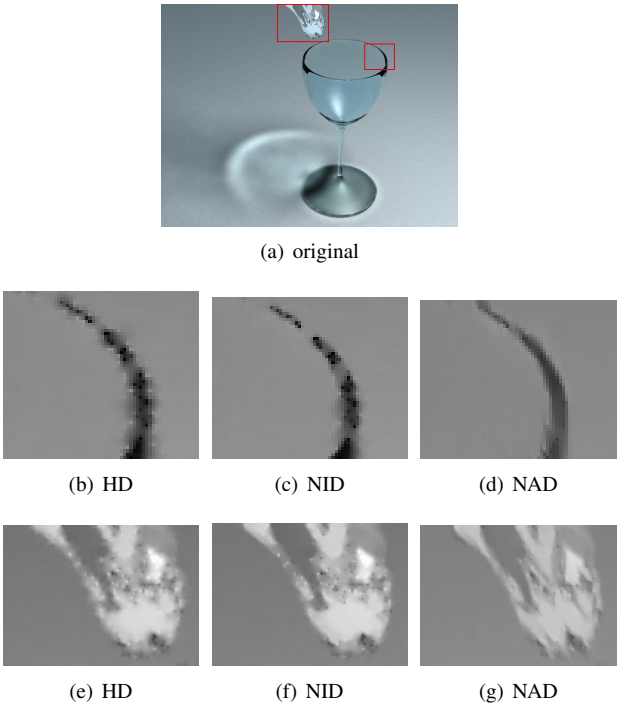
where the diffusion tensor D_{u_σ} can be one of the three choices in order of increasing complexity

- homogeneous diffusion (HD),
- nonlinear isotropic diffusion (NID), or
- nonlinear anisotropic diffusion (NAD).

Examples of reconstructions are shown in Fig. 2. Homogeneous diffusion has a tendency to smoothen edges in the images, but leads to the least costly algorithm. The nonlinear variants attempt to preserve edges better by adjusting the diffusion tensor to the local image features. The NAD regularizer is currently state of the art, as it is best in preserving edges, but is the computationally most expensive one.

The three color channels of an RGB image are encoded separately and solving an equation is necessary for each of them. Typically, a frame rate of about 25 frames per second (FPS) is necessary to achieve smooth real-time playback.

The PDEVC-player is a typical multi-threaded application: One thread interprets the video file and sets up the necessary data structures in main memory. Multiple decompressor threads produce one video frame at a time by solving the associated PDE(s) approximately. Another thread is responsible for displaying. Two ring-buffers are necessary to synchronize the data flow.



(a) original

(b) HD (c) NID (d) NAD

(e) HD (f) NID (g) NAD

Figure 2. Comparing the three different kinds of diffusion.

In the CBEA-optimized version of the player, the decompressor threads off-load the numerical work to an associated SPE. ω-red-black Gauss-Seidel (ωRBGS) solvers are used for the HD and NID regularizers, and a damped Jacobi (JAC) for NAD. More complex solvers, like multigrid methods that are typically used for these types of PDEs, give only small improvement due to the high density of landmarks. Especially JAC is suitable for processing in SIMD, but care must be taken to preserve landmarks where known pixels are given. This is achieved by first calculating a whole SIMD vector containing four new single precision results, regardless of the pixel types. The final result, that will be written back to the Local Storage, is created by selecting from the previous and updated values depending on a bit field describing the landmarks in the current frame. The SPU ISA allows for performing this very efficiently. The kernels are implemented using intrinsics, because the compiler failed in vectorizing and unrolling the loops automatically.

For the image sizes investigated, data from multiple image rows can be held in a LS, so that blocking techniques reduce the DMA transfer with main memory drastically. The ωRBGS solvers perform a whole iteration, JAC two iterations per sweep as described in [8]. Table 1 shows the frame rates that are achievable on a Sony Playstation™ 3 when all six available SPEs are used. These values do not include the bandwidth and effort of the PPE for reading the file and setting up the necessary data structures.

The ωRBGS implementations use the same approach as for preserving landmarks to update only every second unknown, so internally twice the computations need to be performed. From the different types of diffusion tensors, HD leads to a simple five-point stencil for the Laplace operator with fixed coefficients and therefore has a low computational density of 6 Flops per iteration and unknown. The NID regularizer is also approximated by a five-point stencil, but the coefficients are recomputed before each update, requiring 29 Flops per update in total. The highest computational density occurs when nonlinar anisotropic NAD-tensors are used, since they result in a nine-point stencil, whose coefficients are updated every second iteration, resulting in 39.5 Flops per update on average.

Only image data needs to be transferred (4 Byte per pixel and color), since coefficients are calculated on-the-fly on the SPEs. Decoding a single frame using one SPE generates about 120 MB main memory traffic per color frame for the examples in the table.

Table 1. Decompression speed of pdevc. Measured for a resolution of 320×240 pixels. 130 iterations of JAC for NAD or 65 ωRBGS iterations for NID and HD with 10% landmarks were used to obtain comparable times.

regularizer	FPS	bandwidth	computation
HD	101	12 GB/s	8.2 GFlop/s
NID	48	5.8 GB/s	18 GFlop/s
NAD	34	4.1 GB/s	36 GFlop/s

It can be seen that only the HD regularizer has extraordinary bandwidth requirements. To interpret the GFlop rates correctly, it should also be noted that many computations actually performed were not accounted for: the NID kernel reaches impressive 42% GFlop/s internally, but most results are discarded due to the SIMD-vectorization of the ωRBGS method or because they are landmarks.

4. Computational fluid dynamics

Computational fluid dynamics (CFD) has a large number of applications in science and engineering. Besides classical Navier-Stokes solvers, lattice Boltzmann methods (LBM) have become an interesting alternative. LBM use an equidistant grid of cells, so-called lattice cells, that interact only with their direct neighbors. However, both approaches are computationally very expensive, and single computers often do not provide the necessary performance to get results in reasonable time. LBM seem to be especially

suitable for the CBEA due to their simple access patterns, higher computational density, and trivial parallelization on shared memory machines.

cellbm[11] is a prototype LBM solver based on [4] that has been designed especially for the CBEA and uses the common D3Q19 BGK [1, 12] collision model. Its main motivation was to explore the feasibility of blood flow simulation with the related problems—e. g. the complex blood vessel structures— while using specialized hardware efficiently. Single precision was used, since only Cell/BE hardware with slow double precision was available during its implementation.

The memory layout is a key to efficiency and good performance. To save memory, the whole domain is divided into so-called patches of $8 \times 8 \times 8$ lattice cells in size, and only patches containing fluid lattices are actually allocated. This allows efficient processing of moderately irregular domains, while providing all premises for good performance on the CBEA.

The layout allows for efficient data movement to and from the SPEs, transfers of multiple 128-Byte-blocks—corresponding to cache lines of the PPE—with natural alignment in both, main and local storage, result in optimal DMA bandwidth. Besides the patch itself, data from outer edges of possible neighbors needs to be fetched. To avoid the related inefficient gather operations, a copy of these faces and lines is reordered and stored contiguously while processing a patch, and can be retrieved from its neighbors in the next time step easily. Using two buffers for these copies, patches can be processed independently and in any order, so the parallelization is trivial and patches can be assigned dynamically to the SPEs using atomic counters.

Patch data is stored in a structure-of-arrays manner, so all computations can be done in a SIMD way with as many 16 Byte vectors being naturally aligned as possible. SPEs must emulate scalar operations by performing SIMD operations and combining the previous SIMD vector and the SIMD vector containing the desired modification, which makes them extraordinary expensive. Furthermore, loading or writing naturally aligned 16 B vectors are the only memory operations to the LS the SPU supports natively; two aligned loads and a so-called shuffle instruction that extracts the relevant part are necessary to emulate an unaligned load.

Branches may lead to long branch miss penalties on the SXUs and are inherently scalar, so the implementation avoids them wherever possible. Conditional computations are vectorized by computing both possible results in SIMD and creating a select mask according to the condition. The resulting SIMD vector is obtained by combining the different variants according to the mask using a select instruction. The SPU ISA provides various operations for efficient mask generation to perform that efficiently.

Table 2 compares performance of a serial lattice Boltzmann implementation written in C running on various processor types and our SIMD-optimized implementation mainly written in SPU assembly language to demonstrate the importance of SIMDization on the SPUs. The typical means of expressing LBM performance is the number of lattice site updates or fluid lattice sizes updates per second (LUP/s and FLUP/s). A single FLUP corresponds to 167 floating point operations in the optimized SPU kernel. The codes purely run from the CPUs' L2 caches or the SPU's LS, respectively. It can be seen that the PPE cannot keep up with a modern server processor, but performance on the SPU is worst due to the huge overhead of performing scalar operations and branches. Advanced compilers may vectorize simpler scalar algorithms, but they cannot employ SIMD in the LBM program yet.

Table 2. Performance of a straight-forward single precision LBM implementation in C on an Intel Xeon 5160 at 3.0 GHz, a standard 3.2 GHz PPE and SPU, compared with the optimized SPU-kernel for an 8^3 fluid lattice cells channel flow.

	straight-forward C			optimized
CPU	Xeon	PPE	SPU	SPU
MFLUP/s	10.2	4.8	2.0	49.0

There are two approaches for coping with the cache-coherent non-uniform memory access (ccNUMA) topology on the IBM QS blades that provide two Cell processors with an attached main memory and a fast interconnect between the processors. The simpler approach is to allocate all data pagewise alternating on both memory locations, so that a SPE on any CPU will access memory through the nearby and the remote memory bus. Distributing half of the patches to each memory location and the proximate SPEs allows for optimizing for NUMA even better.

Table 3 shows the performance of the whole LBM solver on a Playstation™ 3 and a QS20 blade with different SPE and CPU utilization. Generally, it can be seen that well optimized kernels are able to saturate the memory bus with half of the SPEs available.

When looking at one or two SPEs running on a single CPU, the Playstation™ 3 gets a slightly better performance. On the QS20, the coherence protocol between the two CPUs leads to a lower bandwidth achievable for a single SPE. Memory benchmarks have shown that this is especially true for DMAs writing to main storage.

Both approaches for exploiting the NUMA architecture when utilizing the second CPU and its memory bus can improve performance significantly with an efficiency of 79% and 93%, respectively. If e. g. four Cell processors might

Table 3. Cell/BE MLUP/s performance for a 96^3 channel flow. MFLUP/s = MLUP/s $\cdot \frac{94^3}{96^3}$.

	PS3	QS 20		
CPUs	one	one	both	both
memory	local	local	interleaved	NUMA-aware
1 SPE/CPU	42	40	73	70
2 SPEs/CPU	81	79	129	136
3 SPEs/CPU	93	107	156	189
4 SPEs/CPU	94	110	166	204
6 SPEs/CPU	95	110	174	205
8 SPEs/CPU	N/A	109	173	200

be connected in the future, the efficiency of the simple approach that distributes data blindly will decrease drastically. For applications like the LB method, that are memory bound and whose work can be distributed easily, manual management of data and its memory locations is worthwhile anyway.

5. Molecular dynamics

Molecular dynamics (MD) is another field where the outstanding numerical performance of the CBEA can be of use. One possibility to solve MD problems with a large number of particles and long-range interactions between those effectively are grid-based methods, which are explained in [9]. These methods require fast linear solvers, e. g. multigrid methods. They can be parallelized on a shared memory system with moderate effort and its high floating-point performance and bandwidth make the CBEA a highly interesting architecture for this class of algorithms.

A common issue in MD is the solution of Poisson's equation on an unbounded domain, i. e. with open boundary conditions. For 3D, this problem can be written as

$$\Delta\Phi(\mathbf{x}) = f(\mathbf{x}),\ \mathbf{x} \in \mathbb{R}^3,$$
$$\text{with } \Phi(\mathbf{x}) \to 0 \text{ for } \|\mathbf{x}\| \to \infty,$$

where $\mathrm{supp}(f) \subset \Omega$ is a bounded subset of $\mathbb{R}^3$. For numerical treatment, the equation is discretized, which leads to the following formulation:

$$\Delta_h\Phi(\mathbf{x}) = f(\mathbf{x}),\ \mathbf{x} \in \{\mathbf{x}|\mathbf{x} = h \cdot \mathbf{z}, \mathbf{z} \in \mathbb{Z}^3\},$$
$$\text{with } \Phi(\mathbf{x}) \to 0 \text{ for } \|\mathbf{x}\| \to \infty,$$

with the discrete Laplace-operator Δ_h and mesh size h. This equation is still an infinite system, what prevents the direct numerical solution. For that reason, the system is reduced to a finite one using a stepwise coarsening and expanding grid hierarchy of l levels ($\mathcal{G}_i, i = 1, \ldots l$) as described in [2]. The expanding and coarsening leads to the fact that the number of grid points is not halved in each dimension from one level to the next one, but decreasing slower (compare to Table 4). The values of Φ on the boundary points of the coarsest grid are calculated as

$$\Phi(\mathbf{x}_\delta) = \frac{1}{4\pi} \iiint_\Omega \frac{f(\mathbf{y})}{\|\mathbf{y} - \mathbf{x}_\delta\|_2} \mathbf{dy},$$

what is discretized to

$$\Phi(\mathbf{z}_\delta) = \frac{h^3}{4\pi} \sum_{\mathbf{z}_{\mathrm{fine}} \in \mathcal{G}_1} \frac{f(\mathbf{z}_{\mathrm{fine}})}{\|\mathbf{z}_{\mathrm{fine}} - \mathbf{z}_\delta\|_2}.$$

This evaluation is only sensible for a small number of boundary points because of its high cost. The solution with a multigrid method is supported by that hierarchical grid structure. From the class of multigrid methods, the Fast Adaptive Composite Grid method (FAC) is used, which restricts the original system equation and not the residual equation.

The FAC was implemented using a Jacobi smoother for pre- and postsmoothing, direct injection for restriction, and linear interpolation for prolongation. The program was parallelized on the Cell/BE using domain decomposition. To enhance the execution speed of the code, several optimization techniques were applied:

- SIMDization of the computation kernels: All the operations such as restriction, smoothing, interpolation and treatment of interfaces use the SPE vector operations.
- Linewise processing of the data using double buffering: Each line is 128-Byte-aligned in the Local Storage and the main memory to utilize full memory bandwidth.
- The interfaces between two grid levels need special considerations and are treated using a ghost layer, which avoids the access to both grids at the same time.

After a smoothing step and before the restriction is done, all threads are synchronized to avoid data inconsistencies.

Tests were performed both on the Playstation™ 3 and on the IBM QS20 for different grid sizes. The memory requirements for some of those are specified in Table 4. Since the results are very similar on the Playstation™ 3 and the QS20, but the QS20 enables more opportunities because of its bigger main memory and more SPEs, only the test runs on the QS20 are considered here. The first tests were run using one Cell processor only. Exemplary for the performance of the adapted computation kernels, the runtime of the Jacobi smoother was analyzed. This has been done using exact in-code timing commands at each synchronization event, i. e. after each iteration of the smoother. The timing results for different numbers of

Table 4. Overview of the four finest grid sizes, total number of levels, and memory requirements of the FAC method.

# grid levels	size in each dim. on level 1	2	3	4	memory [MB]
8	35	35	35	23	8
12	67	67	39	35	26
16	131	131	131	71	159
20	195	195	103	99	504

Table 5. Runtimes (in msecs) for one Jacobi iteration depending on grid size and number of threads.

problem size	64^3	128^3	192^3	256^3
1 SPE	1.56	10.1	31.2	70.6
2 SPEs	0.78	5.03	15.6	35.3
3 SPEs	0.63	3.36	10.5	23.6
4 SPEs	0.46	2.55	8.14	17.9
5 SPEs	0.39	2.17	7.25	15.8
6 SPEs	0.34	1.96	6.28	13.8
7 SPEs	0.32	1.81	6.40	13.5
8 SPEs	0.25	1.78	6.08	13.8

unknowns are shown in Table 5.

The question of interest is, whether the memory bandwidth or the floating-point performance is the limiting factor in terms of speed. The first can roughly be computed by $P_{\text{mem}} = \frac{(size-1)^3 \cdot 20}{time}$, as 20 Byte have to be transferred per inner grid point, while the latter is given as $P_{\text{flop}} = \frac{(size-1)^3 \cdot 10}{time}$, since 10 numerical operations are executed per inner grid point. Fig. 3 shows both measures for the previous test runs.

The performance of the Jacobi smoother is basically bound by the memory bandwidth. For up to six SPE threads, scaling of speed is almost ideal, for seven and eight there is hardly any effect, since the memory bus is already saturated. The highest measured value is 22.7 GiB/s.

Additionally, experiments were performed on the QS20 distributing the threads to both processors and an interleaved memory strategy. This strategy allocates memory pages alternating to the two memory buses. So twice the memory bandwidth compared to the default strategy is possible in theory. Practically an improvement of up to 26.8% is gained, as shown in Table 6. The outcome of an advanced memory strategy increases with the number of active SPEs, i. e. for future setups with more processors, exploiting the NUMA architecture more diligently will be crucial.

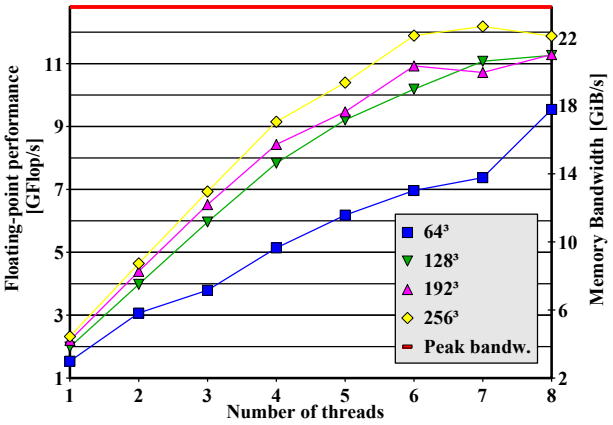


Figure 3. Floating-point performance and memory bandwidth of Jacobi smoother on the QS20.

Table 6. Memory throughput of the Jacobi smoother for grid size 192^3 when using one or both memory buses in GiB/s.

memory strategy	one bus	interleaved	relative speedup
1 SPE	4.1	4.1	0%
2 SPEs	8.18	8.20	0.24%
3 SPEs	12.1	12.2	0.25%
4 SPEs	15.7	16.2	3.1%
5 SPEs	17.6	20.4	15.5%
6 SPEs	20.4	23.0	13.0%
7 SPEs	20.0	25.3	26.8%
8 SPEs	21.0	26.7	26.8%

6. Conclusions

We have demonstrated the potential of the CBEA for several scientific applications and shown that bandwidth and computational power near to the theoretical peak performance is achievable. However, big efforts are necessary to accomplish that. The complexity is only partially caused by specific features of this architecture.

Splitting the task into smaller subtasks and handling synchronization and communication between multiple agents becomes increasingly important since the advent of multicore systems. Heterogeneous architectures only increase complexity in the way that a subtask must fit the abilities of the core type it is executed on.

SIMD is a concept that is very common today, as it is the most efficient way to exploit wide buses and data level parallelism without much complicating the control

logic. The SPU ISA consequently makes SIMD the default case and adds another penalty to performing scalar operations. Similarly, data alignment influences performance on all advanced platforms. Alignment of scalar and SIMD data in memory is restricted on most platforms, or result in decreased performance if not appropriate. However, the discrepancy of performing well aligned SIMD and badly aligned scalar operations on an SPU is unmatched.

The concept of Local Storage, that is managed by copying data to and from main memory via asynchronous DMAs, is perhaps the only concept not met in common general purpose architectures at all. It allows for covering long main memory latencies exceptionally well without using increasingly complex out-of-order cores. On the downside, exact knowledge of the working set and its management is necessary, not mentioning the complexity of distributed, parallel modifications of it. An analogy found on standard cache-based architectures might be the necessary overview of the current working set when using cache blocking techniques, but there it affects only performance and is only relevant for hot spots.

The question remains how much performance can be preserved if one switches to higher-level programming approaches to increase productivity. Since the emphasis of all projects was on how much performance is feasible, this will have to be examined in the future. There is no doubt that libraries and frameworks can ease communication, data partition and movement. But as all general approaches rely on established high-level language compilers, the problem of optimizing numerical kernels in computationally bound applications can be expected to remain.

References

[1] P. Bhatnagar, E. Gross, and M. Krook. A Model for Collision Processes in Gases. I. Small Amplitude Processes in Charged and Neutral One-Component Systems. *Phys. Rev.*, 94(3):511–525, 1954.

[2] M. Bolten. Hierarchical grid coarsening for the solution of the poisson equation in free space. *Electronic Transactions on Numerical Analysis*, 29:70–80, 2008.

[3] I. Galic, J. Weickert, M. Welk, A. Bruhn, A. Belyaev, and H. Seidel. Towards PDE-based image compression. In *Proceedings of variational, geometric, and level set methods in computer vision*, Lecture Notes in Computer Science, pages 37–48. Springer-Verlag, Berlin, Heidelberg, New York, 2005.

[4] J. Götz. Simulation of bloodflow in aneurysms using the Lattice Boltzmann method and an adapted data structure. Technical Report 06-6, Department of Computer Science 10 (System Simulation), Friedrich-Alexander-Universität Erlangen-Nürnberg, Germany, 2006.

[5] IBM. *Cell Broadband Engine Architecture*, Oct. 2007.

[6] IBM. *Cell Broadband Engine Programming Tutorial*, Oct. 2007.

[7] IBM. *Cell BE Programming Handbook Including PowerXCell 8i*, May 2008.

[8] M. Kowarschik. *Data Locality Optimizations for Iterative Numerical Algorithms and Cellular Automata on Hierarchical Memory Architectures*. PhD thesis, Friedrich-Alexander-Universität Erlangen-Nürnberg, Germany, Jun 2004. Advances in Simulation 13.

[9] M. Griebel, S. Knabek, S. Zumbusch, S. Caglar. *Numerische Simulation in der Molekulardynamik*. Springer, 2003.

[10] P. Münch and H. Köstler. Videocoding using a variational approach for decompression. Technical Report 07-1, Department of Computer Science 10 (System Simulation), Friedrich-Alexander-Universität Erlangen-Nürnberg, Germany, 2007.

[11] M. Stürmer, J. Götz, G. Richter, A. Dörfler, and U. Rüde. Fluid Flow Simulation on the Cell Broadband Engine using the Lattice Boltzmann Method. Accepted for publication in the proceedings of the Fourth International Conference for Mesocscopic Methods in Engineering and Science, 2007.

[12] S. Succi. *The Lattice Boltzmann Equation - For Fluid Dynamics and Beyond*. Clarendon Press, 2001.

A Realtime Ray Casting System for Voxel Streams on the Cell Broadband Engine

Valentin Fuetterling
Fraunhofer ITWM
Email: valentin.fuetterling@itwm.fraunhofer.de

Carsten Lojewski
Fraunhofer ITWM
Email: carsten.lojewski@itwm.fraunhofer.de

***Abstract*—In this paper we introduce a volume rendering system designed for the Cell Broadband Engine that only requires a minimum of two voxel slices at a time to perform image synthesis of a volume data set for a fixed number of user defined views. This allows rendering volume data in a streaming fashion and makes it possible to overlap rendering with data acquisition.**

Imagine a screening line at the airport where luggage is examined with an x-ray machine. As luggage passes through the scanner multiple x-ray samples are taken from which 2D voxel slices are derived. These finally form a full volume data set that needs to be displayed for quick analysis. Traditional software volume rendering systems are impractical for such a task as they require the volume data set to be fully available for image synthesis and thus need to wait until the x-raying process has finished.

Our solution is better suited for the depicted situation and related problems as it is able to perform time-critical rendering in parallel with volume generation.

I. Introduction

Volume visualization requires information to be extracted from a 3D scalar field to form a single color value that can be displayed. This mapping can be performed by a maximum/average intensity projection or by evaluating the Volume Rendering Integral [10] which in its discretized form can be computed iteratively with the over operator [12]. In practice the 3D scalar field usually is represented by a uniform grid that is sampled multiple times in order to compute the Volume Rendering Integral or other mappings for every pixel of a viewing plane. Methods that can be used for the sampling process are described in section II. The sampling rate necessary to achieve acceptable results is determined by the Nyquist-Shannon sampling theorem [13]. A huge number of samples need to be taken to visualize a data set resolution of 512^3 voxel or higher which makes volume rendering a compute intensive task. Optimization strategies exist [2], however most of them rely on a pre-process that requires the full volume data set to be analyzed prior to rendering. Thus these strategies are not applicable to volume data sets that are not fully existent at the beginning of the image synthesis process.

Another reason that favors a brute-force approach is its constant runtime characteristic considering a constant volume set resolution as execution time does not depend on the actual volume data which is changing frequently. This is a property often required in our targeted area of application. Using a brute-force solution special purpose hardware is easily designed and very efficient so it is commonly used in todays time critical systems as depicted in the abstract. However special purpose hardware is expensive and inflexible by nature.

We will show that our flexible software approach tailored to the hardware architecture of the Cell Broadband Engine (CBE) is capable of rendering an emerging volume data set 'just in time' from arbitrary view directions and thus delivers the necessary performance for real-time volume data inspection.

II. Volume Sampling Methods

For volume sampling object-order, image-order and hybrid methods exist.

Texture slicing is a popular object-order method for interactive volume visualization. Voxel slices are mapped to polygons that are transformed by projection and blended together in correct order to form the final image [11]. By design this method produces low quality images and requires three sets of voxel slices, each orthogonal to one of the major axes.

A widely used hybrid method is the shear-warp algorithm. Voxel slices are first sheared and projected onto an intermediate plane aligned with the volume which is finally warped onto the viewing plane [8]. The image quality suffers due to the warp process and three sets of voxel slices are required for this technique as well.

As both methods introduced so far demand the full volume data set to be available they are obviously impractical for voxel streaming. An image-order method that does not share this handicap and provides high quality images is ray casting. For each pixel of the view plane a ray is cast into the volume and multiple samples are evaluated along the ray [7]. As each ray can be processed independently this algorithm is very flexible. The streaming model we will introduce in the course of this paper depends on a flexible and easy sampling method in order to be efficient so we decided for ray casting.

III. Cell Broadband Engine

The CBE comprises one Power Processing Element (PPE) and eight Synergy Processing Elements (SPE) which are attached to the Element Interconnect Bus (EIB) together with a Memory Interface Controller (MIC) that provides Direct Memory Access (DMA). The PPE is comparable to an ordinary PowerPC and can offload compute intensive tasks to the SPEs. Each SPE features a Memory Flow Controller (MFC) for data transfer and communictaion, a Local Store (LS) which is 256KB in size and a unified register set with 128 registers.

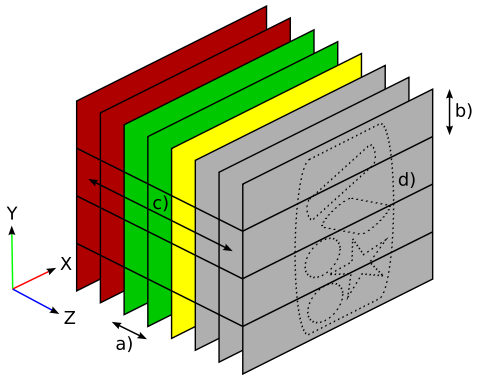


Fig. 1. The two green slices form the slab a) that is currently being sampled, the red slices have already been processed and are no longer available, the yellow slice is being generated by the scanning device and the slices in grey will be produced in the near future. The arrows b) and c) indicate a sub-slice and a sub-slice level respectively. The drawing d) symbolizes the object being scanned.

```
struct ray_packet{
   vec_float4 red, green, blue, alpha; //blending
   vec_float4 cur_traversal_depth;
   vec_float4 dx, dy, dz; //ray direction components
};

struct list_header{
   char num_list_elements;
   char num_ray_packets;
   char view_index; //index for view origin
   char flags;
};

struct list_element{
   unsigned short reserved; //not used
   unsigned short size; // # of ray packets * 128b
   unsigned int lea; //transfer start address
};
```

Fig. 2. Data structures. One list element (8 byte) can reference up to 255 ray packets (128 bytes each).

Each register is 128 bit wide and has SIMD capabilities similar to the AltiVec ISA of the PPE (for more general information on the CBE see [4]).

Communication between the PPE and SPEs can be accomplished by a mailbox mechanism provided by the MFC that allocates an in-tray for each SPE where 32 bit messages can be written to by the PPE or other SPEs. The in-trays work like FIFO queues with a capacity of four messages. A SPE can check its in-tray for new messages at any time. If no new messages are available it can stall until the next message arrives.

In order to process a chunk of data a SPE must initiate an asynchronous DMA transfer to fetch it from main memory into its LS. When more than one continuous chunk of data is required an efficient DMA list transfer can be initiated. The DMA controller utilizes a list to gather multiple chunks from main memory into the LS. The list can also be used to scatter LS data back to memory. Lists must be located in the SPEs LS and each list element must consist of 8 bytes providing the chunk's effective address in main memory and its size in bytes (figure 2).

A DMA transfer will always transmit at least one cache line of 128 bytes. Thus bandwidth is maximized by using addresses and transfer sizes that are a multiple of 128 bytes.

IV. Streaming Model

The Streaming Model is based on the observation that for rendering a volume data set all sampling positions for each ray remain constant if the view and the volume data set resolution do not change. However the actual volume content can be altered arbitrarily as it does not affect the sampling positions. We will refer to such a combination of a constant volume set resolution and a number of constant views from which the volume set is rendered as a *configuration*.

A. Voxel Streams

A Voxel Stream of a volume data set consecutively delivers packets of voxel data that are ordered in respect to time so that their relative positions within the volume data set are known in advance. In practice one packet amounts to one voxel slice and the packet ordering in time is equivalent to the slice ordering along a major axis of the volume data set. From now on we will assume this axis to be the z-axis (figure 1). A voxel stream of a volume data set can be easily rendered using a related *configuration*. For each ray the information which voxel slices are required and when these voxel slices will be available can be precomputed and used at runtime for efficient sampling. The direction of a ray along the z-axis determines whether it traverses the volume front-to-back or back-to-front. For both cases compositing methods exist to compute the Volume Rendering Integral along the ray [12].

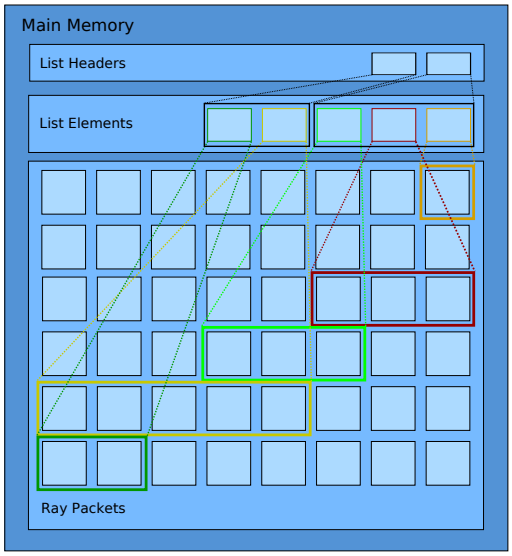


Fig. 3. Illustration of the memory reference hierarchy. The list headers point to continuous blocks of list elements which in turn reference continuous blocks of ray packets inside the ray buffer. The data structures are described in figure 2. List headers are sent to the SPE as mails from the PPE. A SPE can then issue two DMA commands, the first one transmitting the transfer-list elements and the second is using the transfer-list to gather the ray packets.

B. Implementation on the CBE

In this section we describe the implementation of the Streaming Model for a single SPE. The extension of this algorithm for parallel execution on multiple SPEs will be the topic of section V.

1) Sampling: Assuming that two neighboring voxel slices are located in the LS of a SPE the set of rays that possess sample points within the slab formed by the slices (figure 1) must be retrieved from main memory, processed and written back. Because the LS size is limited this ray set must be split into several subsets. A triple buffer approach is necessary to overlap the data transfer with computation. While one set of rays is being processed, the next set is being fetched to the LS and results from the previous set are written back to main memory. As the set of rays associated with one slab of voxel slices can be precomputed, it is possible to generate transfer lists for a given slab. Assisted by a transfer-list the DMA controller can automatically gather a ray set from main memory freeing the SPU for other tasks. The same transfer-list can be used to scatter the ray set back to main memory after the computation has finished. It should be noted here that the transfer lists also minimize bandwidth requirements as no redundant ray data has to be transmitted. Admittedly a transfer list needs to be fetched from main memory prior to execution. However this overhead is insignificant compared to ray data size (figure 2). In order to exploit the SPE's SIMD capabilities rays can be processed in packets of four. Each ray requires three direction components, a traversal depth value and four color components (RGBA) for blending (figure 2). Using single precision floating point values the size of a ray packet amounts to 128 bytes which matches exactly one cache line. Thus ray packets that are discontinuously distributed in main memory do not decrease bandwidth if they share the same set.

Until now we have assumed that two full voxel slices can reside in the LS at the same time. For an appropriate volume resolution however the size of the LS is far too small. For this reason voxel slices need to be partitioned into sub-slices along one axis. We have chosen the y-axis for the remaining of the paper. The partitioning of the volume data set into multiple sub-slice levels is depicted in figure 1. Instead of tracing one full voxel slab at once the process is serialized into sub-slabs with their associated sub-sets of rays. The execution order of the sub-slabs is critical for ensuring correct results as ray sub-sets are not disjoint in most cases. An example is given in figure 4. The ray sub-sets for the sub-slabs A,B,C and D are shown. The sub-set of A is empty, so we do not consider it any further. B, C and D all contain one independent ray that is not shared with any other sub-set (rays 6, 4, 2 respectively). B and C share ray 5 while D and C share ray 3. This sharing implies that C must be processed prior to B and D in order to maintain correctness because the blending of the samples is not commutative[1]. In contrast rays 2, 4 and 6 can be processed in arbitrary order. As the arrows a) and b) indicate dependencies between two sub-sets only exist in one direction of the y-axis. The y-coordinate of the view point (red dot) separates the sub-slabs with potentially positive dependencies (C,D) and potentially negative dependencies (C,B,A). Note that rays with a large y-direction component can share more than two sub-sets. Care must be taken to prevent read-after-write hazards for rays belonging to multiple sub-sets that can arise during DMA transfers.

[1]For the Volume Rendering Integral blending is not commutative. For the maximum/average intensity projection it is.

Multiple ray buffers offer the possibility to circumvent the strict ordering rules for dependent ray sets and to efficiently eliminate read-after-write hazards for the cost of higher memory consumption. Figure 3 illustrates a ray buffer that consists of all the ray packets for a given *configuration*. If copies of this ray buffer are available for all sub-slice levels intermediate blending results can be computed for each level independently from the others. The final compositing of these intermediate results is described in section IV-B4. Multiple ray buffers are even more attractive for the parallelized version of our algorithm (see section V).

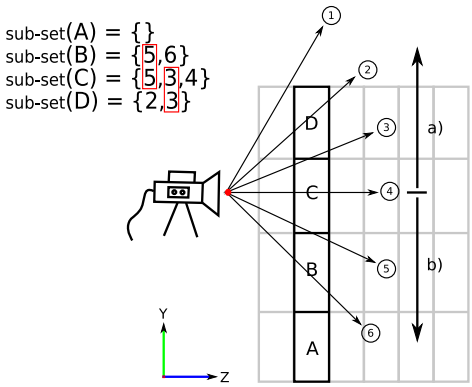


Fig. 4. Dependencies between sub-slabs. The sub-slabs are denoted with upper case letters from A-D. The red dot represents the view origin. The sub-sets of active rays for the sub-slabs are shown in the list, dependencies are marked with red boxes. Independent rays are 1,2,4 and 6 that pierce only one or no sub-slabs. Rays 3 and 5 are positive and negative dependent along the y-axis respectively. The arrows a) and b) indicate the directions of dependence.

So far we have neglected the issue of synchronizing the rendering process with volume acquisition. This is a fundamental requirement in applications like the one depicted in the abstract. We use the mailbox mechanism of the CBE to control the rendering process. When a new voxel slice has arrived in main memory the PPE will send a message to the SPE which contains the list header (figure 2) that allows the SPE to fetch the correct voxel data and ray data (figure 3) into its LS. Every time a SPE has finished processing a ray sub-set it queries its mail in-box for new jobs. If no mail is available it will stall until new work or a termination signal arrives. For a better understanding of how the previously described algorithm is implemented on the SPU side see figure 5.

2) Multiple Views: An obvious approach to rendering multiple views of a *configuration* simultaneously is to utilize one SPE for each view as it is described in section V-B. A different

```
Ptr transferListDataBuffer[3];
Ptr rayPacketDataBuffer[3];
Ptr subslabDataBuffer[2];

Var curMail, nextMail, curIdx, nextIdx, slabIdx;

while( !ExitSignal(curMail) ){

  nextMail = GetNextMail();
  StartTransferListDataGather(nextIdx);
  StartRayPacketDataGather(nextIdx);

  if( SubslabSignal(curMail) ){
    WaitForSubslabDataGather();
    StartSubslabDataGather(slabIdx);
    slabIdx ^= 1; //idx = [0,1]
  }

  WaitForRayPacketDataGather(curIdx);

  for(i=0; i<NumberOfRayPackets(curMail); i++)
    SampleRayPacket(curIdx, i);

  StartRayPacketDataScatter(curIdx);
  curIdx = nextIdx;
  curMail = nextMail;
  nextIdx = (1<<nextIdx) & 3; //idx = [0,1,2]
}
```

Fig. 5. A closer look at the SPU kernel. An actual implementation of this pseudo code can be found in appendix A. *curIdx* and *nextIdx* are indices for the transfer list and ray packet triple buffers. *slabIdx* is the index for the volume data double buffer. *curMail* and *nextMail* contain the list headers received from the PPU. All DMA transfers (except the subslab data gather) are issued with a fence and a tag id equal to the index of the destination buffer.

technique takes advantage of the memory reference hierarchy (figure 3). List headers for multiple views can easily be mixed without the notice of the SPE kernel (figure 6). All information required on the SPE side is a list of all view origins of a *configuration* that can be indexed with the view number contained in a given list header (figure 2). For the parallelized version of our algorithm this approach allows for overlapping certain stalls (section V-A).

3) Preprocessing: Preprocessing for a given *configuration* is straightforward. For each sub-slab all the ray packets of the different views are tested for sample points within the sub-slab to find the valid ray set. The ray packets in the ray set are grouped by continuous main memory addresses (figure 3) and each group is referenced by one list element or more if the group is larger than 16Kb. List elements of the same sub-slab and the same view are combined to form a transfer list that is referenced with a transfer list header (figure 2).

4) Image compositing: Ultimately an image in RGB format is required to be displayed on a monitor. Mapping the ray packets' blending values (figure 2) to pixel colors is straightforward. The red, green and blue color components need to be scaled, cast to integers and stored into the framebuffer. This task can be computed by the PPE or distributed among the SPEs. If multiple ray buffers are used the different blending values for the same ray packet need to be composited first in the correct order. The ray buffers of positive and negative dependent sub-slice levels demand ordering along their respective direction of dependence, starting with the sub-slice level

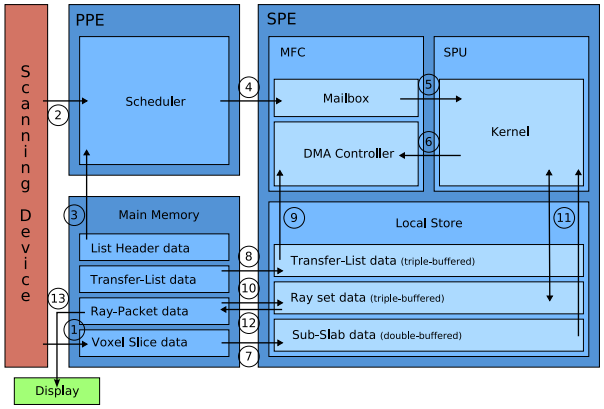


Fig. 6. The data flow of our algorithm. The scanning device writes a new voxel slice into main memory (1) and notifies the PPE (2). The PPE retrieves the next list header (3) and sends it to the SPE's MFC (4). The SPU receives the list header (5) and initiates three DMA transfers (6). The first transmits the transfer-list to the LS (8). The second uses the transfer-list (9) to gather the ray set data into the LS (10). The third moves the voxel data into the LS (7). When all transfers have completed the SPU loads the required data for the sampling process. After computation it writes the results back to the LS (11). As soon as all ray packets have been processed another DMA transfer is initiated (6) that scatters the ray set data back to main memory (12) assisted by the transfer-list (9). Finally an RGB image is extracted from the ray packet data and sent to the display (13).

that contains the y-origin of the given view (figure 4).

At this point all components required for our rendering system have been described to enable an implementation on a single SPE. A summary of the data flow is given in figure 6. Note that for simplicity an unlimited LS size is assumed so that all required data fits into it at the same time. In practise some of the depicted steps need to be subdived into smaller data packages. In the next section we will examine possibilities for distributing our algorithm among multiple SPEs.

V. Parallelization

There are two basic approaches to parallelize our algorithm introduced in section 2. The fine grained solution operates at sub-slab granularity where each SPE is assigned one sub-slice level. The coarse grained model ties one or more independent views to different SPEs.

A. Fine-grained Parallelization

The subdivision of voxel slices into sub-slices which initially has been introduced to account for the limited LS size now offers a convenient approach for parallelization. The sub-slice levels (see figure 1) can be distributed evenly among the participating SPEs for parallel rendering. Each SPE receives only the transfer lists required for the sub-slice levels it processes. Difficulties occur when rays belong to multiple sets for the same slice as this results in dependencies between the different sub-slice levels (see section IV-B1 and figure 4). Sampling order must be preserved for these sets which can be accomplished by the PPE through the mailbox mechanism. The PPE will send the list header of a list containing dependent rays to a SPE only after the dependent rays have been processed by the SPE(s) responsible for the relevant sub-slice

levels. As the ordering can introduce stalls while one SPE has to wait for another to complete its task it is important to carefully generate and schedule jobs during the pre-process to minimize stalls. Ray sets can be split into dependent and independent parts. This allows for execution of independent jobs while a dependent job has to wait. Another source for independent ray sets is available if multiple views need to be rendered (see section IV-B2).

An alternative to preserving ordering among multiple sub-slice levels has already been proposed in section IV-B1. If one ray buffer is dedicated to each SPE rendering can happen in parallel without any constraints. In a final post-process the ray buffers are composited to form a single image as described in IV-B4. However more memory is required for this method.

B. Coarse-grained Parallelization

In case multiple views need to be rendered from the same volume data set, a much simpler approach is to schedule each SPE for a different view. As there are no dependencies between the views the rendering process is equivalent to the one described in section IV-B. The drawback of this method is reduced flexibility as the number of views determines the number of active SPEs. Further on more data needs to be transfered because each SPE requires all sub-slices during the image synthesis process.

VI. Results

The results presented in this section have been measured on three different platforms. The first is a IBM qs20 blade which provides two CBE chips with a clock rate of 3.2 GHZ and 2x512 MB XDR-DRAM. The second is the more up-to-date IBM qs22 blade. In contrast to its predecessor it offers 2x4 GB DDR2-SDRAM and an advanced Double Precision Floating Point Unit which is not utilized by our implementation. As a cheaper alternative results are also reported for a Playstation 3 (PS3) which features one CBE chip clocked at 3.2 Ghz and 256 MB XDR DRAM. However only six SPEs are activated for user applications on the PS3. All processors are running Linux as the operating system.

The volume data set used for rendering is retrieved from a x-rayed backpack (see figure 7) that represents a typical item at an airport screening line. The slice resolution is 512^2 voxels and the slice quantity is 373. For the performance measurements of varying slice resolutions and quantities empty volume data sets are used that contain only zeros. This introduces no implications as one characteristic of our algorithm is that its execution is independent of the actual volume data.

For all measurements we use the fine-grained parallelization technique. During experiments we found that sharing a single ray buffer with all SPEs is inferior in performance to the multiple ray buffer approach by a factor of 3-6. This unacceptable slow-down occurs due to synchronization efforts and serial execution forced by sub-slice dependencies. The advantage of using eight SPEs is therefore diminished. In contrast the increased memory footprint of multiple ray buffers is acceptable as even a screen resolution of 1024^2 fits well into the limited main memory of the PS3. Thus we will focus on the multiple ray buffer approach in the subsequent results.

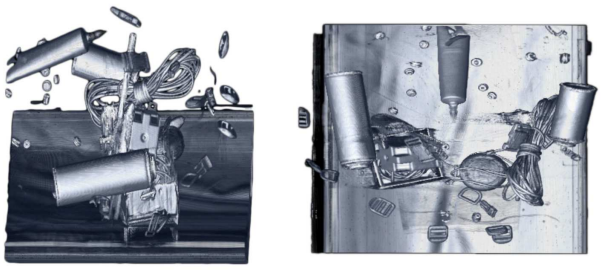


Fig. 7. The backpack data set rendered from different views using a transfer function that clearly shows up internal items.

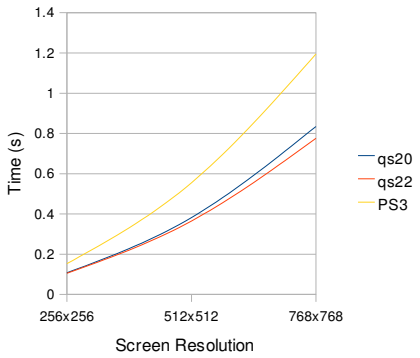


Fig. 8. Timings for the backpack volume on the qs20, the qs22 and the PS3. Only one CBE chip is activated.

Figure 8 shows the performance of our program for different screen resolutions on the qs20, qs22 and the PS3 with only one CBE chip activated. The scaling of the rendering time across different screen resolutions is slightly sub-linear. As the number of ray packets increases, screen resolution independent cost decreases per ray packet. These costs include DMA transfer of voxel slices and setup of the voxel slices on the SPE side. Also the ratio of the number of DMA calls to the number of transfered ray packets is reduced because more ray packets of the same ray set are continuous in main memory. Comparing the PS3, qs20 and qs22 rendering times differ about 10% if performance is normalized to one SPE. Slight differences between the hardware and the OSes might be responsible for this small discrepancy. It is an indicator that our application is not bandwidth limited as the PS3 offers more bandwidth per SPE than the qs20 and qs22.

Figure 9 shows the bandwidth requirements for different image resolutions. While the DMA-put bandwidth requirements remain approximately constant increasingly more DMA-get bandwidth is necessary for smaller resolutions. This phenomenon is related to the ratio of computation to volume data size. As the image resolution decreases less computation has to be performed. However the size of the volume data that

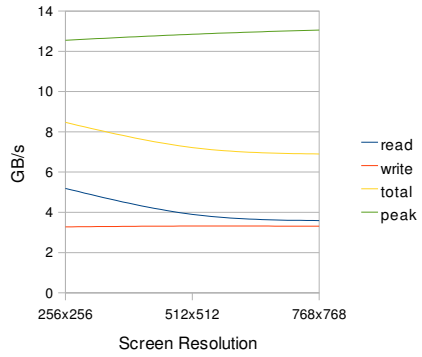


Fig. 9. Bandwidth requirements for the backpack data set measured on the qs22. The graph labled ”peak” demonstrates the maximum bandwidth achieved without rendering computations.

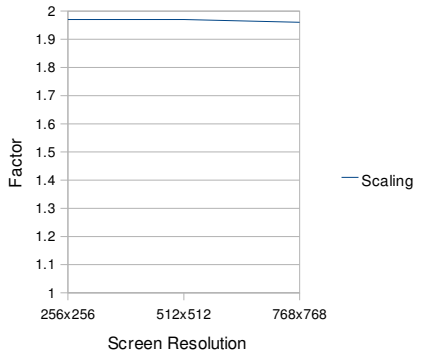


Fig. 10. Performance scaling for both CBE chips of the qs22 with NUMA support.

Slice resolution	Factor	# of ray packets
128x128	0.502	3.2M
256x256	1.00	7.5M
512x512	1.476	11.1M
768x768	1.485	11.3M

TABLE I
PERFORMANCE FOR DIFFERENT SLICE RESOLUTIONS EXPRESSED AS A FACTOR OF THE BASE SLICE RESOLUTION (256^2). THE NUMBER OF PROCESSED RAY PACKETS DURING RENDERING IS SHOWN AS WELL. RESULTS MEASURED ON QS22.

needs to be transfered from main memory to the SPEs does not change because the volume data set is streamed exactly once to the SPEs, regardless of screen resolution. The graph labled ”peak” in figure 9 demonstrate the maximum bandwidth achieved with our application if rendering computations are disabled. This maximum bandwidth verifies that the volume rendering process is not bandwidth limited. The peak bandwidth of the CBE to main memory is around 25 GB/s which is considerably more than the maximum bandwidth delivered by our application. The reason is that the average transfer size of DMA transfers is only around 1 KB for which a reduction in peak performance analogous to our observation is reported by [5]. This also explains the slight increase in maximum bandwidth for larger image resolutions as more coherent ray packets in main memory tend to form larger transfer lists.

In order to efficiently facilitate both CBE chips of the qs20 and qs22 our application provides NUMA support. Due to the highly parallel nature of our algorithm only local memory needs to be accessed during the rendering process. Just for the compositing a small number of remote memory accesses are required. Figure 10 demonstrates almost linear scaling for the backpack example.

Although the timings in figure 8 are not comparable to those achieved by current real-time volume renderering systems [1] they are real-time in the sense that rendering is overlapped with volume acquisition. Many of these real-time volume renderering systems rely on pre-computed acceleration structures that need to be updated or rebuilt when the volume data set changes. Such pre-computation often requires several seconds [9] which is not necessary in our system.

Slice quantity	Factor	# of ray packets
128	0.529	6.0M
256	1.00	11.1M
512	1.438	15.9M
768	1.443	16.1M

TABLE II
PERFORMANCE FOR DIFFERENT SLICE QUANTITIES EXPRESSED AS A FACTOR OF THE BASE SLICE QUANTITY (256). THE NUMBER OF PROCESSED RAY PACKETS DURING RENDERING IS ALSO REPORTED. RESULTS OBTAINED FROM QS22.

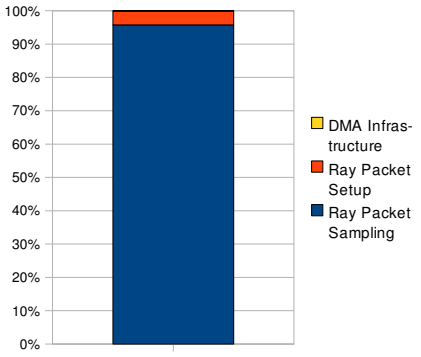


Fig. 11. Runtime distribution for different sections of our algorithm.

Figure 11 presents an evaluation of the runtime distribution for different sections of our algorithm. For the analysis we define three parts: DMA infrastructure, ray packet setup and ray packet sampling. The source code provided in appendix A implements the DMA infrastructure. Less than 1% of the total computation time is consumed by the DMA infrastructure. This indicates that our system of overlapped DMA transfers works very well. More than 95% of the runtime is available for

volume rendering without any main memory latencies. The ray packet setup requires around 4%. This overhead is introduced by the design of our system because ray packets need to be set up repeatedly.

Table I presents the change in performance for different slice resolutions (x- and y-resolution of the volume data set). The screen resolution is 512^2 and the slice quantity (or z-resolution) is 256 throughout. The performance of the results is expressed as a factor of the base result which is obtained from a 256^2 slice resolution. The changes in performance are suprising at first, especially the comparison between the 512^2 and 768^2 slice resolutions. However if the total number of ray packets processed during rendering is considered an almost linear scaling is revealed among the results of table I. It is obvious that the number of processed ray packets can not increase linearly with slice resolution. Changing the slice resolution will not influence the required number of samples at all if a ray is parallel to the z-axis of the volume data set. Analogously if a ray is parallel to the diagonal of a voxel slice it will only scale linearly with the square root of the slice resolution. Most rays lie between both extremes. Additionally our rendering system moves the view away from the volume data set until its projection fits completely onto the view plane. If the resolution is quite different for the three axes of the volume data set this can lead to more inactive ray packets that do not hit the volume at all. This is the case for the 768^2 slice resolution. Table II is analogous to table I. Instead of changing the slice resolution the slice quantity is varied. The slice resolution is constant at 512^2. The results are similar to table I. Performance does not scale with the slice quantity but with the number of packets that need to be processed during rendering. The relationship between slice quantity and number of ray packets is analogous to the relationship between slice resolution and number of ray packets discussed previously.

VII. Conclusion and Future Work

In this paper we have contributed a novel rendering system for time critical volume data inspection that allows for overlapping rendering with volume data acquisition. We have shown how the algorithm can be mapped directly to the hardware features of the Cell Broadband Engine and thus can deliver the performance required for todays real-world applications. Such applications comprise security scans, assembly inspection, medical imaging and others.

Future work should focus on integrating our system into an industrial environment in order to evaluate its suitability for daily use. Additional features like complete local illumination models [6] and multi-dimensional transfer functions [3] could be implemented to further improve image quality.

Acknowledgment

The authors would like to thank the Fraunhofer Institute for Industrial Mathematics (ITWM) for funding and supporting this work.

References

[1] A. Ghosh, P. Prabhu, A. Kaufmann, and K. Mueller. Hardware assisted multichannel volume rendering. In *Computer Graphics International*, pages 2–7, July 2003.

[2] S. Grimm, S. Bruckner, A. Kanitsar, and E. Grolle. Memory efficient acceleration structures and techniques for cpu-based volume raycasting of large data. Symposium on Visualization, 2004.

[3] G. K. J. Kniss and C. Hansen. Multidimensional transfer functions for interactive volume rendering. *IEEE Transactions on Visualization and Computer Graphics*, 8(3):150–162, July–September 2002.

[4] J. Kahle, M. Day, and H. Hofstee. Introduction to the cell multiprocessor. *IBM Journal of RD*, 49(4), 2005.

[5] M. Kistler, M. Perrone, and F. Petrini. Cell multiprocessor communication network: Built for speed. *IEEE Micro*, 26(3):10–23, May–Jun 2006.

[6] J. Kniss, S. Premo, C. Hansen, P. Shirley, and A. McPherson. A model for volume lighting and modeling. *IEEE Transactions on Visualization and Computer Graphics*, 9(2):150–162, 2003.

[7] M. Levoy. Efficient ray tracing of volume data. *ACM Transactions on Graphics*, 9(3):245–261, July 1990.

[8] M. Levoy and P. Lacroute. Fast volume rendering using a shear-warp factorization of the viewing transformation. In *Proc. SIGGRAPH 94*, pages 451–458, July 1994.

[9] S. Lim and B.-S. Shin. A distance template for octree traversal in cpu-based volume ray casting. In *Visual Comput*, volume 24, pages 229–237, 2008.

[10] N. Max. Optical models for direct volume rendering. *IEEE Transactions on Visualization and Computer Graphics*, 1(2):99–108, June 1995.

[11] K.-S. Oh and C.-S. Jeong. Acceleration technique for volume rendering using 2d texture based ray plane casting on gpu. *International Conference on Computational Intelligence and Security*, 2:1755–1758, November 2006.

[12] T. Porter and T. Duff. Compositing digital images. *Computer Graphics*, 18(3):253–259, July 1984.

[13] C. E. Shannon. Communication in the presence of noise. In *Proc. IRE*, 1949.

Appendix

A. SPU Kernel Code Listing

The C/C++ source code provided here is taken from our working implementation. For reasons of insufficient space the initialization code and the the actual volume sampling code had to be omitted.

```
#define SLICE_BUFFER_SIZE 32*1024
#define RAY_PACKET_BUFFER_SIZE 16*1024
#define LIST_HEADER_BUFFER_SIZE 1024

#define MAIL_NUM_LISTELEM(a) spu_add( spu_and( a, 0
    x000000ff ), 1 )
#define MAIL_NUM_PACKET(a) spu_and( spu_rlmask( a, -8 ), 0
    x000000ff )
#define MAIL_INDEX_CAM(a) spu_and( spu_rlmask( a, -16 ), 0
    x000000ff )
#define MAIL_NEXT_SLICE(a) spu_rlmask( spu_and( a, 0
    x01000000 ), -24 )
#define MAIL_NEXT_SUBSLICE(a) spu_and( a, 0x02000000 )
#define MAIL_STOP_RENDERING(a) spu_and( a, 0x04000000 )

class VolumeViewer{

  uchar sliceBuffer[2][SLICE_BUFFER_SIZE] __attribute__ ((
      aligned (128)));
  uchar rayPacketBuffer[3][RAY_PACKET_BUFFER_SIZE]
      __attribute__ ((aligned (128)));
  uchar listHeaderBuffer[3][LIST_HEADER_BUFFER_SIZE]
      __attribute__ ((aligned (128)));
  vec_float4 transferTable[256] __attribute__ ((aligned
      (128)));
  SpuData data __attribute__ ((aligned (128)));

public:

  void render( ullong ea );
```

```
};


void VolumeViewer::render( ullong ea ){

  //no space for all init code
  vec_uint4 startY, sliceY, offsetY, offsetZ, dimX,
       subsliceIndex, subOffsetLUT, curBufferIndex, curMail,
        ppuListElementOffset, VEC_MAXSLICEINDEX,
       VEC_SLICEBUFFER_SIZE, VEC_SLICEBUFFERPART_SIZE,
       VEC_SLICE_SIZE, VEC_SUBSLICE_SIZE, curSlabAdr;
  vec_float4 bMinX, bMaxX, VEC_FLT_MIN;
  uint dBufferIdx, nextSlabIndex, ppuSliceBufferOffset,
       subsliceTransferSize;

  while( !spu_extract(MAIL_STOP_RENDERING(curMail), 0) ){

    //get new header, it will be processed in the next
        iteration
    const vec_uint4 nextMail = spu_splats(spu_read_in_mbox
        ());

    //extract the camera index for the current iteration
        from the previous mail
    const vec_uint4 curCamIndex = MAIL_INDEX_CAM(nextMail);

    //determine buffer index for the next iteration
    const vec_uint4 nextBufferIndex = spu_and( spu_sl(
        spu_splats((uint)1), spu_extract(curBufferIndex, 0
        )), 3);

    //load the origin of the current camera
    const vec_float4 orgX = data.originX[spu_extract(
        curCamIndex, 0)];
    const vec_float4 orgY = data.originY[spu_extract(
        curCamIndex, 0)];
    const vec_float4 orgZ = data.originZ[spu_extract(
        curCamIndex, 0)];

    //extract number of list elements for the next
        iteration
    const vec_uint4 nextListElemNum = spu_sl(
        MAIL_NUM_LISTELEM(nextMail), 3);
    const vec_uint4 nextListTransferSize = spu_andc(spu_add
        (nextListElemNum, 8), spu_splats((uint)0x0000000f)
        );

    //compute address of the current ray packet buffer
    uchar *curRayPacketBuffer = &rayPacketBuffer[
        spu_extract(curBufferIndex, 0)][0];

    //queue dma transfer of list elements
    mfc_getf((void*)&listInfoBuffer[spu_extract(
        nextBufferIndex, 0)][0], data.listInfoBuffer+
        spu_extract(ppuListElementOffset, 0), spu_extract(
        nextListTransferSize, 0), spu_extract(
        nextBufferIndex, 0), 0, 0);

    //extract number of packets for the current iteration
    const vec_uint4 num_packet = spu_sl(MAIL_NUM_PACKET(
        curMail), 7);

    //queue list dma transfer of ray packets for the next
        iteration
    mfc_getlf((void*)&rayPacketBuffer[spu_extract(
        nextBufferIndex, 0)][0], data.rayPacketBuffer, (
        mfc_list_element_t*)&listInfoBuffer[spu_extract(
        nextBufferIndex, 0)][0], spu_extract(
        nextListElemNum, 0), spu_extract(nextBufferIndex,
        0), 0, 0);

    //increase by the number of list elements transfered
    ppuListElementOffset = spu_add(ppuListElementOffset,
        nextListTransferSize);

    if( MAIL_NEXT_SUBSLICE(curMail) ){

      //extract current y-dim offset
      offsetY = spu_splats(spu_extract(spu_slqwbyte(data.
          boundY, spu_extract(subsliceIndex, 0)), 0));
      //next subsliceIndex, wraps back to 0 if
          MAXSLICEINDEX has been reached
      const vec_uint4 msi_mask = spu_cmpeq(subsliceIndex,
          VEC_MAXSLICEINDEX);
      subsliceIndex = spu_andc(spu_add(subsliceIndex, 4),
          msi_mask);

      //local store address of slab data for this iteration
      curSlabAdr = spu_splats((uint)&sliceBuffer[0][0]+
          nextSlabIndex);
      //local store address to store slab data for next
          iteration
      nextSlabIndex ^= SLICE_BUFFER_SIZE;
      const uint nextSlabAdr = (uint)&sliceBuffer[0][0]+
          nextSlabIndex;

      //main memory offset for slice data
      ppuSliceBufferOffset += data.slice_size & spu_extract
          (msi_mask, 0);
      //increase the z offset if the appropriate mail flag
          is set
      offsetZ = spu_add(offsetZ, MAIL_NEXT_SLICE(curMail));

      //transfer slab data for next iteration
      const ullong offset = data.sliceBuffer+
          ppuSliceBufferOffset+spu_extract(spu_slqwbyte(
          subOffsetLUT, spu_extract(subsliceIndex, 0)), 0);
      mfc_get( (void*)nextSlabAdr, offset,
          subsliceTransferSize, 4, 0, 0);
      mfc_get( (void*)(nextSlabAdr+SLICE_BUFFER_SIZE/2),
          data.slice_size+offset, subsliceTransferSize, 4,
          0, 0);

      //compute subslab bounding box
      const vec_float4 bMinY = spu_add(spu_convtf( offsetY,
           0), VEC_FLT_MIN);
      const vec_float4 bMaxY = spu_convtf(spu_add( offsetY,
           sliceY), 0);
      const vec_float4 boundZ = spu_convtf(offsetZ, 0);
    }

    //wait for dma transfers to the current buffer to
        finish
    mfc_write_tag_mask( 1<<spu_extract( curBufferIndex, 0 )
        );
    mfc_read_tag_status_all();

    for( int i=0; __builtin_expect( i<=spu_extract(
        num_packet, 0 ), 1 ); i+=128 ){

      //no space for ray packet setup and ray packet
          sampling code

    }

    const vec_uint4 curListElemNum = spu_sl(
        MAIL_NUM_LISTELEM(curMail), 3 );

    //scatter back intermidiate results
    mfc_putl((void*)curRayPacketBuffer, data.
        rayPacketBuffer, (mfc_list_element*)&listInfoBuffer
        [spu_extract(curBufferIndex, 0)][0], spu_extract(
        curListElemNum, 0), spu_extract(curBufferIndex, 0),
         0, 0);

    curBufferIndex = nextBufferIndex;
    curMail = nextMail;
  }

  //...
}
```

Comparison of High-Speed Ray Casting on GPU using CUDA and OpenGL

Andreas Weinlich, Benjamin Keck, Holger Scherl, Markus Kowarschik and Joachim Hornegger

***Abstract*—Iterative 3D volume reconstruction is one of the most compute- and memory-intensive applications in the field of medical image processing. The iterative reconstruction consists of two major compute intensive steps: Forward- and back-projection. Both steps have to be applied repeatedly in each iteration and several iterations are necessary until a reconstruction result with high image quality is available. As a consequence iterative reconstruction techniques are rarely used in practical CT-like systems. To step towards clinical usage it is mandatory to apply highly parallelized low-cost processing architectures such as the stream processors on current GPUs (Graphics Processing Units). In order to achieve high image quality we implemented the forward-projection using a volumetric ray cast method. We have carefully adapted our implementation to two recent GPU-programming tools, CUDA (NVIDIA Compute Unified Device Architecture) and OpenGL (Open Graphics Language). In terms of execution performance and implementation complexity we compared both tools for the forward-projection step.**

***Index Terms*—computed tomography, iterative reconstruction, volumetric ray casting, CUDA, OpenGL, forward-projection**

I. Introduction

For the last years mostly analytical methods like the filtered back-projection have been used in clinical Cone-beam CT (Computed Tomography) systems in order to achieve 3D volume reconstructions out of acquired 2D projection images. Iterative 3D reconstruction algorithms like SART (Simultaneous Algebraic Reconstruction Technique) or SIRT (Simultaneous Iterative Reconstruction Technique) [1] can produce less reconstruction artifacts [2], i.e. reconstructions using a small amount of projections, even though they are much more time consuming than the conventional Feldkamp algorithm [3]. The iterative reconstruction consists of two major compute- and memory-intensive parts: A forward- and a back-projection step. We recently showed a comparison of latest acceleration technologies for the back-projection step [4]. Especially ray-driven implementations of the forward-projection like a volume ray caster, which are used for their superior precision [5], suffer from their computational demand. Also in other application domains ray casting algorithms are extensively used, like in the field of 2D-3D registration [6]. To overcome the limitations and build real time solutions for clinical application, it is necessary to use hardware architectures with massively parallel computation capabilities. Like in similar applications, one of the most appropriate and cost efficient solutions are modern graphics cards [7]. For example, NVIDIA's GeForce 8800 GTX and QuadroFX 5600, which we utilized for our tests, use 128 stream processors in parallel and can additionally benefit from some hardware-accelerated features like texture interpolation. Recently NVIDIA has developed a C-like general purpose API for these GPUs to implement for example parallelized numerical algorithms.

Unfortunately, the first CUDA versions up to 1.1 had still some drawbacks like missing support for 3D textures. This feature was introduced in the recently published major release, CUDA 2.0. But maybe still the compiler is not as sophisticated as in the OpenGL graphics programming language. Furthermore, as a matter of principle, it can only be used on modern NVIDIA graphics cards. On the other hand there exists another very interesting hardware platform for CUDA applications called NVIDIA Tesla. In this paper we compare highly optimized implementations of ray casting using CUDA 1.1, CUDA 2.0 and OpenGL regarding programming techniques, implementation time, and execution performance.

A. Weinlich, B. Keck and J. Hornegger are with the Friedrich-Alexander-University Erlangen-Nuremberg, Department of Computer Science, Chair of Pattern Recognition (LME), Martensstr. 3, D-91058 Erlangen, Germany.

H.Scherl and M. Kowarschik are with Siemens Healthcare, CV, Medical Electronics & Imaging Solutions, P.O.Box 3260, D-91050 Erlangen, Germany.

II. Related Work

In the medical field, perspective projections are often used to simulate and approximate the physical process of X-ray attenuation. Over two decades ago, Joseph [8] introduced an improved algorithm for forward-projecting rays. His algorithm is not as precise as a ray cast based algorithm, but less computationally complex, which was more important at this time. Later Xu et. al. compared popular interpolation and integration methods for use in CT [5] and showed that a ray cast based algorithm is comparable to the other superior methods regarding the root mean square (RMS) error. Because modern GPUs provide hardware-accelerated interpolation, we decided to implement the forward-projection using ray casting.

The iterative reconstruction performance of graphics accelerators has often been evaluated using OpenGL and shading languages [7], [9].

III. Methods

In this section we describe the principle of the forward-projection step. Second, we explain our CUDA-based and OpenGL-based implementations.

A. Forward-projection

We use a volumetric ray casting approach for the forward-projection step. Its basic functionality is shown in Figure 1 and the algorithm is shown in Algorithm 1. To determine the grey level value of a certain pixel on the image plane, a straight

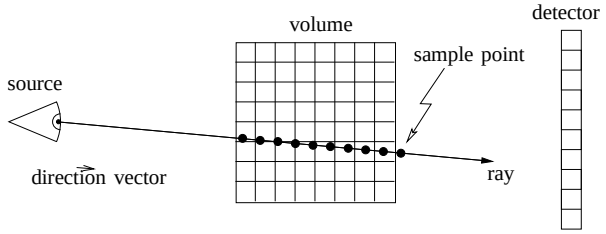


Fig. 1. Ray casting principle.

line ("ray") is drawn pointing from the optical center towards the pixel position. Afterwards voxel intensity values inside the cuboid are sampled equidistantly along the ray. These sampling values add up to the desired gray level value in the image. As a result we get a perspective projection of the volume data.

Algorithm 1 Forward-projection with a ray casting algorithm

```
for all projections do
  compute source position out of projection matrix
  compute inverted projection matrix
  for all rays inside the projection do
    compute ray direction depending on the image plane
    normalize direction vector
    //RAY CASTING
    compute entrance and exit point of the ray to the cuboid
    if ray hits the cuboid then
      set sample point to the entrance point
      initialize the pixel value
      while sample point is inside the cuboid do
        add up the computed sample value at current
        position to the pixel value
        compute new sample point for given step size
      end while
    else
      set pixel value to zero
    end if
    normalize pixel value to world coordinate system units
  end for
end for
```

The physical process of acquiring an X-ray image works just as well. In particular, in this case the optical center depicts the X-ray source whereas the image plane depicts the detector. While Strobel et. al. [10] have shown that the image quality of a reconstruction can be improved by using projection matrices instead of assuming an ideal geometry, we decided to use this parameterization in our implementation.

Furthermore this section describes some general features that are common to both implementations, CUDA as well as OpenGL. There are some different methods to get the direction vector of the ray, which is the first step in the inner for loop in Algorithm 1. A simple one is to take two position vectors, compute the difference vector, and normalize it. Such positions are the optical center, the 3D coordinate of the pixel position, or the points where the ray enters or leaves the cuboid. For example the position of the optical center can be obtained

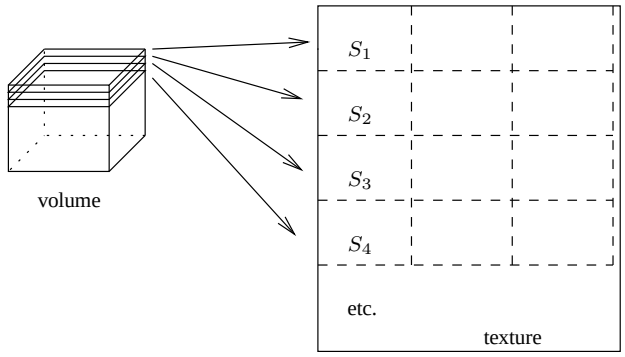


Fig. 2. Volume representation in a 2D texture by Slices S_i.

from the homogeneous projection matrix which is designed to project a 3D point to the image plane. Depending on the output format of the projection (2D image- vs. 3D world-coordinates), this matrix has three or four rows. In the latter case, the vector can be found in the fourth column of the inverted matrix (first three components). In the case of a 3×4 matrix it is possible to drop the fourth column, invert the 3×3 matrix and multiply the inverse with the previously dropped fourth column to get the center position. This holds, because in case of a perspective projection with projection matrices, this fourth column depicts the shift of the optical center to the origin of the coordinate system. But due to the fact that this translation occurs not before the rest of the transformations, these have to be undone in multiplying the inverse. Galigekere et. al. have shown already how to reproject using projection matrices in [11].

In the next step the entrance position of the ray into the volume has to be calculated. The used method to get the entering and leaving points depends on the implementation. Between those points the cube is equidistantly sampled. To get one sampling position, we take the entry vector and add the direction vector multiplied with the step size times a counter variable. The following sampling step itself proves to be crucial for the algorithm's efficiency. In order to get satisfying results, a sub-pixel sampling is required, which introduces a trilinear interpolation.

For a realistic simulation of X-ray imaging, the Beer-Lambert law has to be fulfilled approximately:

$$I = I_0 \cdot e^{-\int_{t(\underline{v}_{\text{source}})}^{t(\underline{v}_{\text{detector}})} \rho(\underline{x}(t))\, dt} \tag{1}$$

The densities p are integrated along the line $\underline{x}(t)$ (or added up in a discrete manner). Afterwards, they are transformed with the exponential-function and multiplied with an initial X-ray intensity to get the target intensity value. This subsequent transformation will not be considered here as it can be computed for example during a post-processing step. For the application in algebraic reconstruction, a pre-processing of the original X-ray images may be also appropriate to fit the ray caster projections.

B. Implementation in CUDA

CUDA offers an easy to use C-like application programming interface with some extensions. There are two different parts in each CUDA implementation: A host part, which executes in a CPU thread, and a device part (kernel), which is invoked by the controlling CPU thread, but runs in parallel on the GPU device. In our case the program instructs the graphics card to create a semi-parallel thread for each ray. On our hardware up to 128 of these threads can be processed in parallel. Most of our CPU code uses CUDA specific API functions for allocating data structures on the device and to transfer data to the graphics memory and back to RAM.

In the kernel code, the inverse of the projection matrix is used to get the ray direction out of the pixel position in the projection image. In order to check whether a sampling position is inside the cuboid, the entrance and exit distances with respect to the optical center are computed. In each step the entrance position is incremented by a step size value until it reaches the exit distance. A critical issue in CUDA 1.1 is the sampling step since it does not provide support for 3D textures. So unfortunately a trilinear hardware interpolation is not available for the CUDA 1.1 API. In consequence, a workaround had to be applied that used just the bilinear interpolation capability of the GPU. It does a successive linear software interpolation in between stacked 2D texture slices (see Figure 2). Therefor, desired values are fetched from proximate stack slices with hardware-accelerated bilinear interpolation. These sampling steps are substituted with only one hardware-accelerated 3D texture fetch in CUDA 2.0 and OpenGL.

C. Implementation in OpenGL

The OpenGL implementation is more tricky in some aspects. This is a consequence of the fact, that OpenGL is intended to be used in graphics applications. Nevertheless there are some similarities like the perspective projection. In the past years, the API itself was made more flexible by means of shader languages, which makes it possible to implement a forward projection using OpenGL [12].

Like in CUDA, the implementation divides into a CPU and a GPU part. The CPU part (OpenGL code) was written in C++. In our implementation the GPU fragment shader program is written in the shader language, GLslang. The OpenGL API invokes this code for each pixel in the projection. Due to the fact that a pixel exactly corresponds to a ray, this threading is the same as in CUDA. However, unlike CUDA, this partitioning can not be defined by the programmer directly. In fact this correspondence is a fixed OpenGL fragment shader feature.

In the OpenGL code, there are some initializations establishing a desktop window for rendering. Furthermore, frame buffer objects are initialized in order to store the projection into a texture. As stated above, the volume data resides in a 3D texture like in CUDA 2.0. This fact allows for the utilization of hardware supported tri-linear interpolation. The projection matrix for an image has to be transformed in order to fit the OpenGL coordinate system. Afterwards some variables are transferred to the shader, the six faces of the cuboid are drawn using vertices, the cuboid is rendered to a texture and finally this texture is copied back to host memory.

During the rendering, the instructions within the shader program are executed instead of the texture lookup. These instructions differ slightly from the corresponding CUDA code. Corners of the 3D texture have been assigned to the corners of the cuboid, so the OpenGL texturing step provides the entrance position of the ray automatically in terms of interpolated texture coordinates. The ray direction vector can be obtained like it was outlined in the last section. In each step the program checks, whether the sampling position is still inside the cuboid. As mentioned, the sampling itself reduces to a simple 3D texture fetch.

	NVIDIA GeForce 8800GTX	NVIDIA QuadroFX 5600
Core clock	575 MHz	600 MHz
Shader clock	1350 MHz	1400 MHz
Memory amount	768 MB	1500 MB
Memory interface	384-bit	384-bit
Memory clock speed	900 MHz	800 MHz
Memory bandwidth	86.4 GB/s	76.8 GB/s

TABLE I
TECHNICAL SPECIFICATION OF BOTH GRAPHICS CARDS USED IN OUR EVALUATION

IV. RESULTS

In order to compare the performance of both approaches, we measured execution times with different test parameters on an NVIDIA GeForce 8800 GTX as well as on an NVIDIA QuadroFX 5600. Even though both graphics cards are assembled with the NVIDIA GPU ”G80” they are slightly different stated in Table I. Our evaluation system is a Fujitsu-Siemens Workstation ”R650” using the Intel 5400 chip set. The graphics cards are connected each via a PCI Express x16 slot.

For measurement purpose we used different projection geometries and volume phantoms. If the phantom fits inside the field of view, there exist rays that do not go through the cuboid at all (case ”far”). These rays consume a minimum of the computation time and the computation finishes noticeably faster compared to the test case where optical center and image plane are close to the cuboid (case ”near”). Associated parameters that have direct impact on the computational complexity of the ray caster are image size (number of pixels and with it number of rays) as well as the sampling rate along one ray (distance of sampling positions compared to the size of a voxel). Due to the fact that in CUDA the execution of the kernel and thus the ordering of the texture fetches can be configured by the block configuration [13], we also compared this parameter for CUDA 1.1 and CUDA 2.0. Large images have some additional side effects. On one hand, they allow a more flexible schedule of threads, on the other hand each ray

Blocksz.	512^2 pixels near	512^2 pixels far	1024^2 pixels near	1024^2 pixels far	2048^2 pixels near	2048^2 pixels far
16×16	48.2	87.7	106	107	409	301
32×8	50.5	101	109	111	412	315
32×16	46.4	113	107	116	411	308
64×4	59.8	127	109	138	424	340
64×8	54.4	129	111	127	415	330
128×2	74.0	132	121	222	425	397
128×4	57.8	124	115	185	431	372
256×1	98.2	140	169	302	449	597
256×2	68.9	124	122	218	448	467
512×1	100	141	167	253	441	593

TABLE II
BLOCK PARAMETER COMPARISON OF RUNTIMES USING CUDA 2.0 ON THE NVIDIA GEFORCE 8800GTX IN SECONDS WITH 400 PROJECTIONS AND DIFFERENT PROJECTION SIZES AT A STEP SIZE OF 0.25 OF THE VOXELSIZE

# Proj.	FoV	512^2 pixels CUDA 1.1	512^2 pixels CUDA 2.0	512^2 pixels OpenGL
1	near	6.22	**1.60**	3.25
	far	6.47	**1.60**	3.24
16	near	14.2	**3.30**	5.32
	far	18.2	**4.97**	6.45
100	near	55.5	**13.1**	21.7
	far	92.5	**24.4**	25.3
400	near	145	**41.8**	47.0
	far	386	**88.7**	90.3

TABLE III
COMPARISON OF RUNTIMES USING THE NVIDIA QUADROFX 5600 IN SECONDS (CUDA 1.1 VS. CUDA 2.0 VS. OPENGL) WITH A PROJECTION SIZE OF 512 SQUARED AND A DIFFERENT NUMBER OF PROJECTIONS AT A STEP SIZE OF 0.25 OF THE VOXELSIZE

needs some initial calculation steps apart from the sampling. Unless otherwise noted, a block consists of 16×16 pixels within the projection. A block parameter comparison for the GeForce 8800 GTX using CUDA 2.0 is shown in Table II. Another important parameter is the number of projections to be acquired from the same volume data. The time required for initialization steps, preparing the data structures and loading the volume data to the device, is spent just once. So, a high number of projections reduces the influence of such preceding computations (e. g. 1.6 seconds for CUDA and 3.2 seconds for OpenGL on the QuadroFX 5600).

If both implementations are well optimized, it is expected that OpenGL will perform better than CUDA 1.1 and comparable to CUDA 2.0.

We use a projection size of 512×512 or 1024×1024 pixels. The resulting execution times for the GeForce 8800 GTX and QuadroFX 5600 using a projection size of 1024×1024 are shown in Table IV and Table V, and for the QuadroFX 5600 in Table III using a projection size of 512×512 and 2048×2048 in Table VI. In Figure IV we give an overview of the dependency on the projection size using the QuadroFX 5600. In order to hit most of the voxels in the volume, the step size (sampling rate) must not be greater than 1 voxel. If we actually do not want to loose information, it should be at most 0.5 of the voxel size. In favor of a smooth projection image a step size of 0.25 voxels would be even better. A direct comparison between GeForce 8800 GTX and QuadroFX 5600 for the computation time depending on the step size is shown in Figure 7. The number of projections that can be computed consecutively depends on the reconstruction algorithm. For example, SART computes only a single projection per volume update. In contrast, SIRT processes all projections consecutively before a volume update is performed in the iteration. Certainly there are algorithms in between such as the ordered subset approach.

In Figure 6 we can see the dependency of the execution time on the chosen step size for most common parameters. The measurements do not include the time required to write-back the projections to the host memory or even to hard disk, because it is not required for a complete GPU implementation of iterative CT reconstruction. Moreover, those times (especially the write back to disk) can be hidden behind the computation of the next slices. For example a projection of 100 images, 1024×1024 including write back takes approximately one additional second on the QuadroFX 5600 (0.35 sec write back to host, 0.19 sec write back to hard disc and 0.54 sec for deletion of data, etc.). In most cases OpenGL and CUDA 2.0 operate two or three times faster than CUDA 1.1. For a small number of projections, the results seem to depend on the other parameters, i.e. the initialization time of the API, which takes longer for OpenGL. In contrast, the tests with 400 projections show a more interesting behavior. The best executed results are highlighted in bold in Table IV, III, V and VI. In Table IV it can be seen that CUDA 2.0 is faster in all tests by a constant offset of approximately 8 seconds on the GeForce 8800 GTX. In Figure the dependency on the step size for the two different geometric setups in a common setting for SIRT (1024×1024 pixels; 400 projections) is shown. The time increases almost linear with the step size except for an offset.

To give an impression of GPUs computational performance we finally compare a specific test case also with a CPU implementation. The CPU implementation is a single-threaded non-optimized straight-forward implementation of the raycast method as stated in Algorithm 1. The program is executed on our test system equipped with two Intel Xeon E5410 processors running at 2.33 GHz. For a simple comparison we used 16 projections 1024×1024 at a step size of 0.25 of the voxel size. Table V proves a performance of 5.16 seconds for such configuration using the "near" field of view setting on the NVIDIA QuadroFX 5600. We measured 764 seconds for the single threaded CPU program. This indicates a maximal speedup factor of 148.

V. DISCUSSION

At higher numbers of projections the execution times for the CUDA implementation which uses 2-D textures to compute

# Proj.	FoV	1024^2 pixels CUDA 1.1	CUDA 2.0	OpenGL
1	near	9.4	**3.8**	12.1
	far	9.4	**3.8**	11.9
16	near	20.6	**7.5**	15.5
	far	27.3	**8.2**	15.5
100	near	86.4	**28.4**	36.4
	far	126	**30.2**	37.3
400	near	299	**107**	115
	far	527	**108**	116

TABLE IV
COMPARISON OF RUNTIMES USING THE NVIDIA GEFORCE 8800GTX IN SECONDS (CUDA 1.1 VS. CUDA 2.0 VS. OPENGL) WITH A PROJECTION SIZE OF 1024 SQUARED AND A DIFFERENT NUMBER OF PROJECTIONS AT A STEP SIZE OF 0.25 OF THE VOXELSIZE

# Proj.	FoV	1024^2 pixels CUDA 1.1	CUDA 2.0	OpenGL
1	near	6.38	**1.60**	3.22
	far	6.71	**1.59**	3.25
16	near	16.2	**5.16**	6.94
	far	21.4	**5.02**	7.09
100	near	70.5	**25.1**	27.4
	far	114	**24.6**	29.5
400	near	245	**99.8**	103
	far	515	**90.9**	109

TABLE V
COMPARISON OF RUNTIMES USING THE NVIDIA QUADROFX 5600 IN SECONDS (CUDA 1.1 VS. CUDA 2.0 VS. OPENGL) WITH A PROJECTION SIZE OF 1024 SQUARED AND A DIFFERENT NUMBER OF PROJECTIONS AT A STEP SIZE OF 0.25 OF THE VOXELSIZE

trilinear interpolations are much longer than for our other implementations using 3D textures (CUDA 2.0 or OpenGL). It is therefore essential to use the hardware-accelerated functions of the GPU in order to optimize the execution performance of our CT reconstruction applications. The constant execution time offset in each test case (approx. 12 seconds in OpenGL and 4 seconds in CUDA 2.0 on the GeForce 8800 GTX) can be explained with the copy process of the volume data to the graphics memory along with some other initializations. With a QuadroFX 5600 card we observed a significantly smaller

# Proj.	FoV	2048^2 pixels CUDA 1.1	CUDA 2.0	OpenGL
1	near	7.70	**1.59**	3.27
	far	7.26	**1.58**	3.26
16	near	37.7	**15.8**	17.7
	far	30.6	**11.4**	13.3
100	near	208	**95.4**	98
	far	173	**67.4**	70.4
400	near	841	**392**	397
	far	864	**284**	290

TABLE VI
COMPARISON OF RUNTIMES USING THE NVIDIA QUADROFX 5600 IN SECONDS (CUDA 1.1 VS. CUDA 2.0 VS. OPENGL) WITH A PROJECTION SIZE OF 2048 SQUARED AND A DIFFERENT NUMBER OF PROJECTIONS AT A STEP SIZE OF 0.25 OF THE VOXELSIZE.

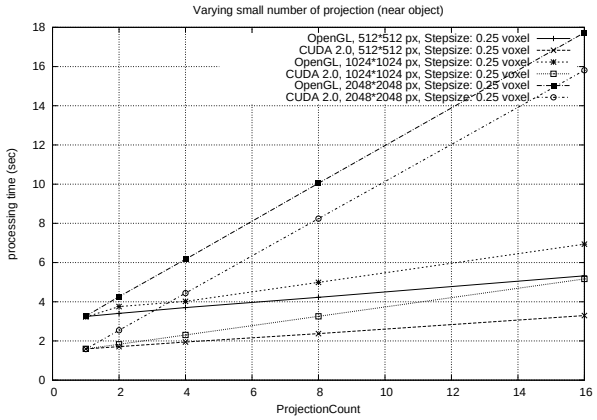


Fig. 3. CUDA 2.0 and OpenGL comparison for varying projection size and a small amount of projections.

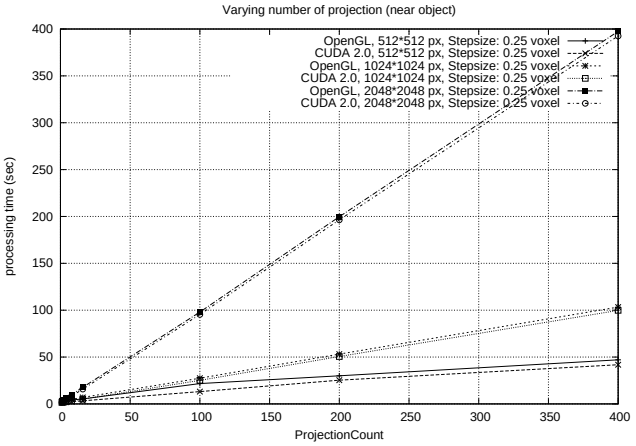


Fig. 4. OpenGL and CUDA 2.0 comparison with similar execution time behavior for varying projection count and size.

difference in initialization time between OpenGL and CUDA. As expected, the increase in runtime is almost linear in the step size and the number of projections. With increasing image sizes, OpenGL and CUDA are able to dispatch the parallel computations more efficiently to the multiprocessors of the GPU up to a certain amount. This is the reason why the execution time increases remarkably slower and does not scale with the number of pixels in an image. Merely at 2048×2048 pixels and an ROI including the complete data, there can be seen a strong increase in execution time. As a consequence, it seems that projection images with 1024×1024 pixels are optimally suited for current GPUs generations. An implementation in OpenGL requires more implementation efforts for non-experts because it was built as a graphics programming language for real-time rendering of vertex-based 3D scenes. In contrast, a ray casting in the C programming language can be more easily ported to CUDA, as it only requires some adaptations for the parallelization strategy. However an OpenGL expert can implement such an algorithm in equivalent time compared to a C-Programmer using CUDA.

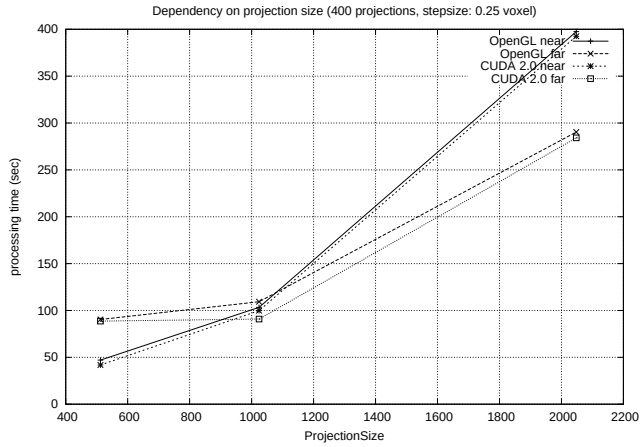


Fig. 5. The projection size dependency on the QuadroFX 5600.

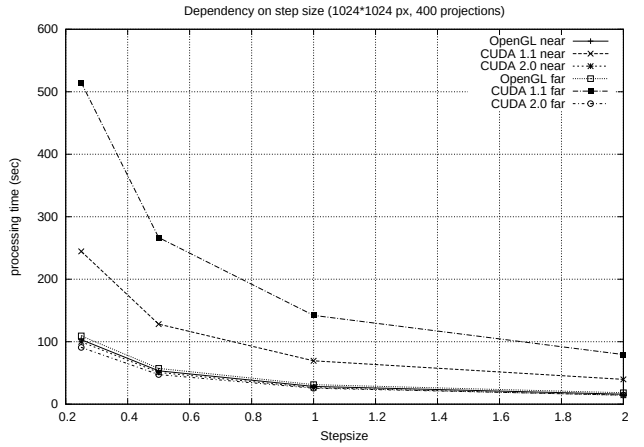


Fig. 6. The stepsize dependency on the QuadroFX 5600.

VI. Conclusion

We have presented three highly optimized implementations of volume ray casting usable i.e. as the forward-projection step in iterative reconstruction. Our comparison of the execution times shows that the performance of the recent CUDA version is even slightly better than an implementation using OpenGL. Older CUDA versions should not be used for ray casting due to the lack of 3D texture support. CUDA unveils the processing power of graphics cards even for programmers that are not specialists in computer graphics. The OpenGL implementation required much more implementation time, however it can also be used with no CUDA capable devices. On the other hand, the Tesla series from NVIDIA can only be used together with CUDA.

Acknowledgments

This work is being supported by Siemens Healthcare, CV, Medical Electronics & Imaging Solutions. We wish to give special thanks to Dr. Klaus Engel who supported us with his wide OpenGL API knowledge.

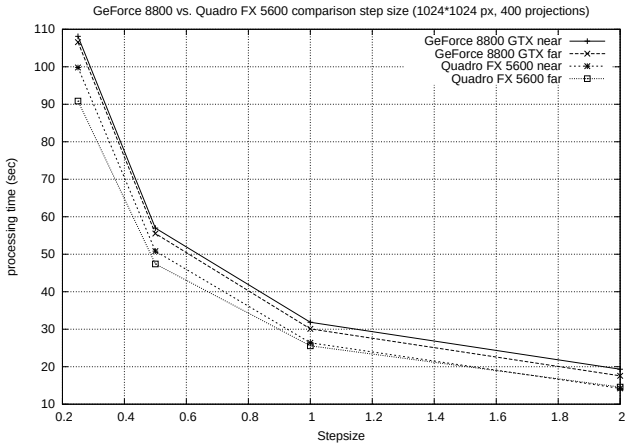


Fig. 7. GeForce 8800 GTX to QuadroFX 5600 comparison on step-size using CUDA 2.0.

References

[1] A. Andersen and A. Kak, "Simultaneous algebraic reconstruction technique (sart): A superior implementation of the art algorithm," *Ultrasonic Imaging*, vol. 6, no. 1, pp. 81–94, January 1984.

[2] K. Mueller and R. Yagel, "Rapid 3d cone-beam reconstruction with the algebraic reconstruction technique (art) by utilizing texture mapping graphics hardware," *Nuclear Science Symposium, 1998. Conference Record. 1998 IEEE*, vol. 3, pp. 1552–1559, 1998.

[3] L. Feldkamp, L. Davis, and J. Kress, "Practical cone-beam algorithm," *Journal of the Optical Society of America*, vol. A1, no. 6, pp. 612–619, 1984.

[4] H. Scherl, B. Keck, M. Kowarschik, and J. Hornegger, "Fast GPU-Based CT Reconstruction using the Common Unified Device Architecture (CUDA)," in *Nuclear Science Symposium, Medical Imaging Conference 2007*, E. C. Frey, Ed., 2007, pp. 4464–4466.

[5] F. Xu and K. Mueller, "A comparative study of popular interpolation and integration methods for use in computed tomography," *Biomedical Imaging: Nano to Macro, 2006. 3rd IEEE International Symposium on*, pp. 1252–1255, April 2006.

[6] A. Kubias, F. Deinzer, T. Feldmann, S. Paulus, D. Paulus, B. Schreiber, and T. Brunner, "2d/3d image registration on the gpu," in *Proceedings of the 7th Open German/Russian Workshop on Pattern Recognition and Image Understanding (OGRW), FGAN-FOM*, Ettlingen, 2007.

[7] K. Mueller, F. Xu, and N. Neophytou, "Why do commodity graphics hardware boards (GPUs) work so well for acceleration of computed tomography?" in *SPIE Electronic Imaging Conference*, San Diego, 2007, (Keynote, Computational Imaging V).

[8] P. M. Joseph, "An improved algorithm for reprojecting rays through pixel images," *IEEE Transactions on Medical Imaging*, vol. MI-1, no. 3, pp. 192–196, 1982.

[9] M. Churchill, "Hardware-accelerated cone-beam reconstruction on a mobile C-arm," in *Proceedings of SPIE*, J. Hsieh and M. Flynn, Eds., vol. 6510, 2007, p. 65105S.

[10] N. K. Strobel, B. Heigl, T. M. Brunner, O. Schuetz, M. M. Mitschke, K. Wiesent, and T. Mertelmeier, "Improving 3D image quality of x-ray C-arm imaging systems by using properly designed pose determination systems for calibrating the projection geometry," in *Medical Imaging 2003: Physics of Medical Imaging. Edited by Yaffe, Martin J.; Antonuk, Larry E. Proceedings of the SPIE, Volume 5030, pp. 943-954 (2003).*, ser. Presented at the Society of Photo-Optical Instrumentation Engineers (SPIE) Conference, M. J. Yaffe and L. E. Antonuk, Eds., vol. 5030, Jun. 2003, pp. 943–954.

[11] D. H. R. Galigekere, K. Wiesent, "Cone-beam reprojection using projection-matrices," *IEEE Transactions on Medical Imaging*, vol. 22, no. 10, pp. 1202–1213, 2003.

[12] K. Müller, "Fast and accurate three-dimensional reconstruction from cone-beam projection data using algebraic methods," Ph.D. dissertation, Department of computer and information science, Ohio State University, Columbus, Ohio, USA, 1998.

[13] N. Corp., "NVIDIA CUDA Compute Unified Device Architecture Programming Guide," 2007. [Online]. Available: http://www.nvidia.com/cuda

RapidMind Stream Processing on the PlayStation 3 for a 3D Chorin-based Navier-Stokes Solver

Vincent Heuveline
Numerical Methods in High Performance Computing
Steinbuch Centre for Computing
Karlsruhe Institute of Technology, Germany
vincent.heuveline@kit.edu

Dimitar Lukarski and Jan-Philipp Weiß
SRG New Frontiers in High Performance Computing
Exploiting Multicore and Coprocessor Technology
Karlsruhe Institute of Technology, Germany
dimitar.lukarski@kit.edu, jan-philipp.weiss@kit.edu

***Abstract*—In Cell's heterogeneous multi-core configuration the issues related to data management are not treated at the hardware level and have to be handled explicitly by the user and its software environment. For dedicated applications the resulting performance gains are impressive and exemplified in the existing literature. However, an extensive programming effort and code redesign are required to port applications to the Cell processor. As a promising alternative, RapidMind's stream processing model provides an easy to use programming approach in single-threaded manner where all data transfers and scheduling are handled implicitly. Our investigation shows that algorithms of a complex fluid dynamic application can be mapped to the Cell BE in an efficient way. But due to programming model- and problem-intrinsic restrictions with respect to temporal locality, main memory resources and memory bandwidth are the limiting factors for a profitable deployment in that field of application.**

***Index Terms*—Cell BE, RapidMind, Navier-Stokes solver, bandwidth-bound algorithms, stencil operation**

I. Introduction and overview

Technological limitations, strong desire for tremendous performance and the market pressure to sell emerging products have lead to the development of multi-core processors as mainstream technology in multimedia and technical computing. However, currently only a limited number of applications can benefit from the huge potential of these new hardware architectures. Extensive efforts are required to fill the gap between hardware capability and software efficiency. A major challenge relies on the ability to design hardware-aware codes for fine-grained parallelism which should be as generic as possible. This situation becomes especially pronounced in the context of the Cell Broadband Engine (BE) jointly developed by Sony, Toshiba and IBM (STI).

Aiming at huge bandwidth and unrivaled performance, the designers of the Cell BE decided to refrain from the classical approach of a hardware-controlled nested cache hierarchy that automatically brings data closer to the cores and cares for increasing temporal locality of frequently used data. For Cell BE the users have to care at the software level for providing the data for the cores in time. As a consequence, programming models for Cell have become more complicated and data management and organization is a critical issue. A common programming approach relies on multi-threading of applications. However, parallelization is a time-consuming, difficult, and error-prone task. A specific expertise for hardware-aware software development is required that cannot always be handled by specialists from diverse scientific disciplines who have to rely on fast computations and reliable numerical simulations.

The RapidMind Multi-core Development Platform is providing a promising approach for sticking with a single-threaded programming style and simultaneous full exploitation of parallel power of emerging multi-core platforms. Based on a stream processing model, RapidMind's solution is conceived to overcome the management of the complicated data transfers to and from the cores. As a consequence, programmers and algorithm designers can concentrate on the details of the algorithm instead of coping with technical difficulties. Moreover, RapidMind is offering a portable solution that may run on the Cell processor, Graphics Processing units (GPUs) and x86 multi-core CPUs.

Our goal in this paper is to investigate the potential of RapidMind's high level concept on the Cell BE for the solution of a highly compute time- and memory-demanding problem in computational fluid dynamics (CFD). As opposed to an approach encompassing only measurements of BLAS 3 routines (like SGEMM or the LINPACK benchmark) or isolated application kernels, the main emphasis of this paper is put on the usability and performance of the considered technology for a CFD problem which meets the needs of typical real world applications. We are investigating if RapidMind's high-level approach can leverage the parallel potential of the Cell BE and if all necessary components are provided to map complex numerical algorithms to Cell's multi-core architecture in an efficient manner with respect to both computational throughput and development time. As we will outline, the stream processing approach conflicts with concepts for increasing temporal locality. Our investigation is mainly performed on Sony's PlayStation 3, Cell's branch into the mass market. Some results are compared to those obtained on IBM Cell BladeCenter QS21.

This work is organized as follows. Section II gives a short list of related work in the field of CFD, stream processing, and implementation work on Cell. In Section III we are providing a description of the fluid dynamic problem under consideration: a three-dimensional incompressible Navier-Stokes solver based on Chorin's projection method and discretization on

staggered grids. Section IV is dedicated to a short overview of the architecture details of the STI Cell BE and its incorporation into PlayStation 3 and the IBM Cell BladeCenter QS21. In Section V we describe the RapidMind stream processing model. Section VI shows the expected performance bounds for the implementation of the considered fluid dynamic model problem. In Section VII we present the performance results of our fluid dynamic solver on PlayStation 3 implemented in RapidMind. We conclude with a summary and outlook in Section VIII.

II. Related work

Investigations of modern numerical methods and applications on the Cell BE and other multi-core platforms employing diverse programming models are subject of current research activities. Performance results for a Lattice Boltzmann based fluid dynamic solver on Cell BE can be found in [1]. Another parallel implementation of a similar fluid dynamic solver on an emerging hardware platform can be found in [2]. To the authors' knowledge there are no performance results for RapidMind implementations on the Cell BE available in the literature. Our work is a first contribution. Related work with RapidMind on GPU for an application of bioinformatics can be found in [3].

III. Fluid dynamic model problem

Fluid dynamic problems are challenging problems in terms of numerical modeling and algorithmic implementation. The model problem under consideration is three-dimensional, time-dependent, and force-driven motion of a viscous fluid in a cube $\Omega = [0,1]^3$ subject to non-slip boundary conditions in the time interval $(0,T]$ described by the incompressible Navier-Stokes equations

$$\begin{aligned} \partial_t \mathbf{v} + (\mathbf{v}\cdot\nabla)\mathbf{v} - \nu\Delta\mathbf{v} + \nabla p &= \mathbf{f} \quad \text{in } \Omega\times(0,T], \\ \nabla\cdot\mathbf{v} &= 0 \quad \text{in } \Omega\times(0,T], \\ \mathbf{v} &= 0 \quad \text{on } \partial\Omega\times(0,T]. \end{aligned}$$

The unknowns are the scalar pressure p and the vector-valued fluid velocity $\mathbf{v}$. The right hand side of the momentum equation is the prescribed force $\mathbf{f}$. In our model scenario a circular force is applied to a fluid subject to zero boundary conditions and zero initial conditions for the velocity, i.e. $\mathbf{v}(t=0)=0$ in Ω.

With Chorin's projection method [4], the time-dependent equations can be solved by an iterative and explicit time stepping scheme. Starting with inititial value $\mathbf{v}^0$, a sequence $\tilde{\mathbf{v}}^k$, p^k, $\mathbf{v}^k$, $k=1,\dots$, is determined by

$$\frac{\tilde{\mathbf{v}}^{k+1}-\mathbf{v}^k}{\triangle t} + (\mathbf{v}^k\cdot\nabla)\mathbf{v}^k - \nu\Delta\mathbf{v}^k = \mathbf{f}^k \quad \text{in } \Omega, \tag{1}$$

$$\Delta p^{k+1} = \frac{1}{\triangle t}\nabla\cdot\tilde{\mathbf{v}}^{k+1} \quad \text{in } \Omega, \tag{2}$$

$$\frac{\mathbf{v}^{k+1}-\tilde{\mathbf{v}}^{k+1}}{\triangle t} = -\nabla p^{k+1} \quad \text{in } \Omega. \tag{3}$$

We refer to [5] for more details regarding the properties and the adequate formulation of the boundary conditions for this scheme. The upper index k is related to the definition of the discrete time t^k, and $\triangle t := t^{k+1} - t^k$ describes the uniform time step size. The explicit character of this scheme leads to stability constraints with respect to the time step size (see e.g. [5] and references therein).

For the spatial discretization we use finite-difference approximations on staggered MAC grids [6] with uniform grid size. The domain is discretized into small cubes with edge length $h = 1/n$. The pressure values are associated with the centers of the cubes whereas the three velocity components in x-, y- and z- direction are associated with the faces of the cubes with normals in corresponding directions. For the Laplacian operator Δ we apply regular 7-point stencils. Central differences are invoked for the divergence and the gradient operator. The stencils for the non-linear term are obviously more involved (see e.g. [7] for more details). A detailed description of the discretization can be found in [8], [9].

The most expensive part of the solution process in terms of necessary operations is the projection step (2) where a linear system of equation (LSE) has to be solved for the discrete pressure. The other two steps are explicit. We consider the conjugate gradient (CG) method [10] for the solution of the LSE. Several iteration steps have to be performed to compute an approximate solution.

IV. Cell Broadband Engine

The Cell Broadband Engine (BE) is a processor architecture relying on innovative concepts for heterogeneous processing and memory management. In the STI approach specialized cores are added to a main unit motivated by increasing both computational power and memory bandwidth.

The main unit of the Cell BE is the Power Processing Element (PPE) running at 3.2 GHz. The PPE mainly acts as controller for the eight Synergistic Processing Elements (SPE) which usually handle the computational workload. The SPEs are SIMD processors with 256 KBytes local, software-controlled memory called Local Store (LS). The LS does not operate like a conventional CPU cache since it does not contain hardware structures or protocols that predict which data may possibly be reused or loaded. The SPEs mainly perform SIMD instructions on 128-bit wide 128 entry register files. Furthermore, they manage data transfers by Direct Memory Access (DMA). Each SPE reaches 25.6 GFlop/s performance for single precision (SP) instructions. In the first version of Cell, double precision (DP) instructions are not fully pipelined leading to modest performance of 1.8 GFlop/s per SPE. In its latest release PowerXCell 8i, DP performance reaches 12.8 GFlop/s. The eight SPEs are connected via the Element Interconnect Bus (EIB) delivering aggregate peak bandwidth of 204 GByte/s. However, the data has to be accessed from main memory connected via a Memory Interface Controller (MIC) to the EIB. Details on the Cell architecture can be found in [11].

In the present work two different test environments are investigated: Sony's PlayStation 3 and IBM's BladeCenter QS21. The PlayStation 3 (PS3) incorporates the Cell BE

processor at a low price. However, only six SPEs are available under a Linux OS due to economic viability and maintenance of the game OS. In PS3 there are only 256 MByte of XDR DRAM main memory. Thereof only around 200 MByte are accessible under Linux. The Cell BladeCenter QS21 is equipped with two Cell processors and each of them can access 1 GByte of XDR DRAM main memory (2 GByte together). Both of the processors on the BladeCenter are connected by the Broadband Engine Interface protocol (BIF) providing around 20 GByte/s bandwidth for data exchange. With only one processor of QS21 running, BIF enables memory access to the second XDR DRAM. The aggregate SP compute capability of the SPEs of QS21 is 409.6 GFlop/s and 153.6 GFlop/s on PS3. On PS3 theoretical peak main memory bandwidth is 25.6 GByte/s. On QS21 both XDR DRAMs can be accessed with 25.6 GFlop/s each. Effects of Non-uniform Memory Access (NUMA) have to be considered.

The main drawback of the Cell architecture is that the Local Store is usually not large enough for the entire application data. Therefore, data must be decomposed into pieces small enough to fit into local memory. These data pieces must be replaced subsequently through the DMA without losing the performance gain associated to the usage of multiple SPEs. Main memory bandwidth turns out to be the bottleneck for many data-intensive applications. On Cell BE there is no full conformity with IEEE 754 norm [12] in SP concerning rounding modes, treatment of denormals and overflow.

With respect to the software-controlled memory hierarchy, explicit data transfers, and different instruction sets of the PPE and SPEs, a sophisticated programming model for the Cell BE is required. A basic approach relies on the Software Development Kit (SDK) [12] provided by IBM. With no attempt toward an exhaustive list, other solutions are CorePy [13], MPI MicroTask [14], Sequoia project from Stanford [15], IBM's Octopiler [16], Barcelona Supercomputing Center's CellSs Superscalar [17], and the Mercury Multi-Core Framework [18]. In our work we are focusing on RapidMind's stream processing approach (see Section V).

V. RapidMind stream processing

In this paper we consider the RapidMind Multi-core Development Platform for the implementation of our fluid dynamic model problem on the Cell BE. RapidMind (RM) is based on a stream processing model [19]. The RM platform is a collection of libraries built on C++ with some language extensions and its own predefined data types. Existing compilers can be used; only specific header files need to be included. The RM platform manages execution of RM programs on the target device (the SPEs on Cell). It handles all memory transfers and load balancing. Programmers can implement their applications in a single-threaded manner with no explicit parallelism and no hardware knowledge. The major advantages are the ease of programming and the portability of implementations. In its current release RM is supporting Cell BE, GPUs from nVIDIA and AMD/ATI, and x86 multi-core CPUs. Anyhow, direct portability remains limited due to architectural differences like size of local memory and different instruction sets. Backend specific tuning options for optimization and different data layout are still necessary.

In the stream processing approach, computation is applied in form of so-called kernels acting on data streams. In the RapidMind framework data streams are tiles of arrays transferred to the SPEs. The kernels represent sets of instructions defining uniform operations in the sense of the Single Program Multiple Data (SPMD) execution model. A basic requirement to reach flexibility in coding is dynamic flow control by loops and branches within the kernels. However, no side effects are allowed due to undetermined order of execution on the data chunks. This concept directly excludes data conflicts and race conditions but comes along with limitations with respect to algorithmic flexibility.

The main disadvantages of stream processing models are the model-intrinsic and design-specific limitations to data reuse. Since the kernels are applied to tiles of large arrays, already loaded data cannot be kept for later reference. Moreover, intermediate write operations of processed data are prohibited. As a consequence, concepts designed for temporal locality cannot be meaningfully applied. Data reuse within several steps of complex algorithms cannot be exploited and some data may have to be transferred several times.

RM comes along with its own data types simplifying data transfers and vectorization on SIMD units. New data types are values (short vectors), arrays (containers for values) and programs (kernels operating on arrays). The most efficient data structure for computations on Cell are 4-component vectors (Value4f for floats) that fit in registers of the SIMD units on the SPEs. Inputs for kernels are either values or arrays of values. For kernels with values as input, the optimal data transfer is managed by RM. For blocking the data transfers for algorithmic purpose (e.g. for stencil computation), array tiles of prescribed size are transferred. In order to reach maximal bandwidth, data should be fetched from main memory in 16 KByte chunks. In- and output arrays are limited to 16 KByte size. This restriction can be bypassed by the definition of several in- or output arrays. Data for the kernels can be fetched via input arrays or into local arrays. Local arrays are supported on Cell but not on GPUs due to the small size of local memory. Input and output arrays need to have the same size but may have different types. Input arrays are automatically double-buffered for overlap of communication and computation. The access to arrays can be controlled by array accessors. Several options are available to manipulate array access patterns. For non-uniform operations on RM's values, these short vectors can be rearranged via permutations (swizzling) or by write-masking. RM programs (kernels) are compiled dynamically by the RM platform to machine language on the target hardware specified by included backends. Compilation is invoked the first time the kernels are used.

As observed in our experiments, access times to RM arrays are longer than accesses to pure C++-arrays. However, they are mandatory for data transfers. Rearrangements of short vectors (values) by swizzling is more costly on Cell compared to

GPUs. A real disadvantage are repeated dynamical compilations of kernels if source code is spread on several files.

All computations in this work are performed by using RM Development Platform Version 2.1. In this version of RM there is no double precision support. Further information about the RM platform can be found in [20], [21], [22]. A different and also promising stream processing approach is provided by CUDA from nVIDIA [23].

VI. Theoretical and expected performance

The performance of numerical algorithms on hardware is mainly influenced by two components: data transfers between a nested memory hierarchy and the compute cores, and computations, mainly performed on floating point operands. Parallel codes additionally suffer from delays due to necessary communication and synchronization between different threads or read-write conflicts in the memory system.

A lower bound for the total run-time T_R of a numerical application is given by $T_R \geq T_C + T_T$, where T_C is the compute time and T_T is the time for the data transfers. In the case of asynchronous transfers and overlapping of communication and computation, the lower bound can be taken as $\max\{T_C, T_T\}$. On a given platform an algorithm is compute-bound for $T_C > T_T$ and bandwidth-bound for $T_T > T_C$. A simple performance model can be derived by knowledge of the algorithm and hardware characteristics. Let f the number of floating point operations (Flop) in the algorithm to be performed and w the number of floating point words to be transferred from memory to the cores and back (if necessary doubly counted). Then we get lower bounds $T_T \geq 4w/B$ for 4 byte single precision words where B (in GByte/s) is the maximal bandwidth between memory and cores. Here, we have to consider the narrowest bottleneck of possibly several data paths like main memory to on-chip/on-board memory or caches to the cores. Furthermore, we find $T_C \geq f/P$ where P (in GFlop/s) is the accumulated theoretical peak performance of the functional units. An upper bound for the effective performance P_{eff} of the corresponding implementation can hence be given by

$$P_{\text{eff}} = \frac{f}{T_R} \leq \frac{f}{\max\{T_C, T_T\}} \leq \min\left\{P, \frac{fB}{4w}\right\}.$$

For unlimited bandwidth (B very large) or compute-dominated algorithms (f very large), the upper bound is basically the peak performance P. For unlimited compute capability (P very large) effective performance is bounded by $P_{\text{eff}} \leq fB/(4w)$. For the PS3 at least 24 Flop (32 Flop for one 8-SPE Cell) have to be performed per data transfer such that the algorithm is not bandwidth-bound. This does not apply to all components of our simulation. Hence, the transfers dominate the total time of computation. For specific problems dedicated strategies for reduction of memory transfers may be developed in order to cope with this issue.

Memory transfer reduction strategies include restriction to single precision with a basic performance gain of a factor of two on most architectures. Matrix operations on regular grids may be reduced to application of fixed stencils preventing matrices to be transferred. Temporal blocking techniques like loop skewing and circular queue [24], [25], [26] give significant benefits for stencil applications but are intrinsically based on explicit solution schemes. It is however important to note that in the case of numerical solution of time-dependent partial differential equations, explicit schemes come along with severe constraints with respect to the time step size which may cancel the achieved benefits.

In the following we investigate the algorithmic aspects more detailed. For our application on PS3, we consider a fluid in a cube with edge length n of size $N = n^3$. On the applied staggered MAC grids the components of the velocity profile corresponding to a single fluid cell are gathered into 3-tuple RM vectors (Value3f). Scalar pressure values of four fluid cells can be gathered into 4-tuple pencils in z-directions (Value4f). Data chunks of maximum 16 KByte size are built by $4 \times 16 \times 16$ 3-tuples (quarter cubes) for the velocity and the force and $16 \times 16 \times 4$ 4-tuples (full cubes) for the pressure.

With respect to the stencil operation we consider the following data layout. The order of the subcubes as well as the subcubes itself are organized in lexicographical order where z is the unit stride direction. Within the subcubes two different data layouts are considered. In the strict lexicographical subordering data from neighboring subcubes have to be fetched in fragmented manner. To overcome this issue, data of the subcubes are reorganized by distinguishing interior nodes and nodes at the interfaces. For efficient memory access within the stencil kernel, the arrays for the subcubes start with the nodes of its six interfaces and are followed by the interior nodes, all in lexicographical order and grouped in 4-tuples. Due to the overlap of the subcubes, the number of total data transfers in the stencil kernel is $2.75N$ for our data layout (instead of $2N$ without overlap).

In the advection step (1), a nonlinear stencil has to be applied to the three velocity components held in 3-tuple vectors of quartered subcubes. For the application of the nonlinear stencil thirteen 3-tuple vectors corresponding to neighboring fluid cells are required. In order to access the components in a non-uniform way, 3-tuples vectors have to be reorganized by swizzling into temporary RM-tuples. Furthermore, an update of the velocity components, computations of the 7-point Laplace stencils for the velocity components, and initialization of the force profile representing the right hand side have to be performed. Moreover, homogeneous Dirichlet boundary conditions apply to the intermediate velocity. The advection step is the kernel with highest computational intensity due to evaluation of the non-linear term in (1).

In the projection step (2), Poisson equation with Neumann boundary conditions has to be solved which results in solution of a sparse and structured LSE where the matrix is represented by a regular 7-point stencil acting on scalar values for the pressure. The solution method considered in this paper relies on the conjugate gradient (CG) method [10]. In order to simplify the analysis of the obtained performance results, we do not include any kind of preconditioning method.

The CG method consists of several components. In each iteration step of the CG method, a single stencil operation corresponding to the matrix-vector multiplication, a scalar product, a vector norm, and three `saxpy` vector updates [27] have to be performed. Due to the Neumann boundary conditions in the projection step (2), the computed pressure is defined only up to a constant value. Therefore, a normalization is required after application of the stencil and after every vector update in order to ensure solvability of the LSE due to perturbations caused by SP rounding errors.

In Table I we illustrate the properties of the components of a single CG iteration step. The first column shows the number of occurrences in a single step. The following columns present the number w of necessary words transferred, the number f of Flop, the computational intensity f/w (in asymptotic value for large N), and the theoretical performance bound for the complete operation. In the practical implementation some steps, e.g. stencil operation and scalar product, can be combined into a single kernel with associated reductions of transfers. Each component of the CG step consists of $O(N)$ operations on $O(N)$ elements. The necessary number of CG steps depends on the condition number of the LSE which is in our case of order $O(n^2) = O(N^{2/3})$. The number of iterations to reach a prescribed error tolerance results in $O(n) = O(N^{1/3})$. The total amount of operations to solve the pressure equation sums up to $O(nN) = O(N^{4/3})$. Memory requirements for the CG method are basically storage of four vectors.

Function	Occ.	w [#words]	f [#Flop]	f/w	Perf. bound [GFlop/s]
Vector norm	1	$N+1$	$2N-1$	2.0	12.8
Scalar product	1	$2N+1$	$2N-1$	1.0	6.4
Vector update	3	$3N+1$	$2N$	0.7	4.2
Stencil	1	$2.75N$	$8N$	2.9	18.6
Normalization	4	$2N$	$2N$	1.0	6.4

TABLE I
COMPUTATIONAL INTENSITY AND PERFORMANCE BOUNDS FOR COMPONENTS OF A CG STEP.

In the velocity update step (3), a pressure gradient has to be added to the velocity components. Due to the memory organization, four data tiles for the velocity and a single data tile for the pressure have to be accessed in each subcube of size 16^3.

The total memory requirements are about $96N$ Bytes ($2 \cdot 9 \cdot 4N$ for two velocity and force vectors with 3 components, $2 \cdot 4N$ for the pressure, $4 \cdot 4N$ for CG). Here, we assumed that data for the velocity, force, and pressure are kept twice: in lexicographical order for visualization and in block lexicographical order for computation. On PS3 we have ≈ 140 MBytes available, allowing for $n = 112$. On QS21 with 2 GByte we can use $n = 272$.

Typical applications for stream processing should have a computational intensity f/w, defined as the ratio of performed Flop per memory transfer, that is polynomially increasing in N. For all components of our applied fluid dynamic solver the number w of data transfers is of order $O(N)$, i.e. in principle

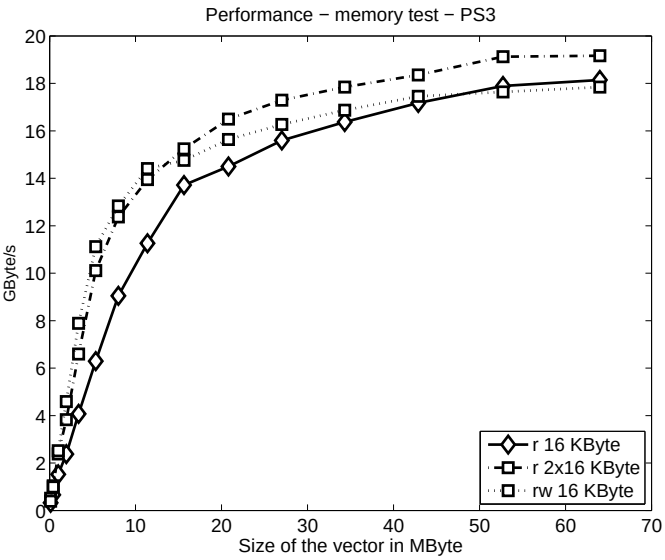


Fig. 1. Main memory bandwidth on PS3.

single vectors are transferred. On every vector component only a couple of computations can be performed. So the total number f of floating point operations in each kernel is also of order $O(N)$. Hence, computational intensity is asymptotically constant with respect to problem size, i.e. $O(1)$. As a consequence, all kernels represent bandwidth-bound operations on Cell for large N. Limiting factor is the still impressive theoretical main memory bandwidth of $B = 25.6$ GByte/s. The outstanding peak performance $P = 153.6$ GFlop/s on PS3 cannot by far be fully utilized. In fact, as our experiments show, effective bandwidth on PS3 is less than expected. By counting the total number of transfers we find that a single CG step of the projection step (2) is more than twice as expensive as compared to the advection step (1) and the velocity update step (3).

VII. IMPLEMENTATION AND EXPERIMENTAL PERFORMANCE RESULTS

As outlined in the previous sections, our Chorin-based Navier Stokes solver is a bandwidth-bound algorithm on the Cell BE. On account of this, we start with an investigation of the main memory performance on PS3. Figure 1 shows that kernels with 16 KByte in- and output arrays reach a bandwidth of approximately 17.8 GByte/s. Bandwidth slightly increases when only read operations are treated. A further increase can be observed for two input arrays of size 16 KByte. In Figure 2 the same investigation is performed on 8 SPEs of one Cell processor of the QS21 with disabled BIF-connection to the second Cell processor and its main memory channel. By default settings on QS21, the BIF connection is enabled giving higher bandwidth as compared to 25.6 GByte/s by employing the second main memory channel. Peak performance is about 20.5 GByte/s for read and write operations and about 24.3 GByte/s considering only read operations of size 16 KByte.

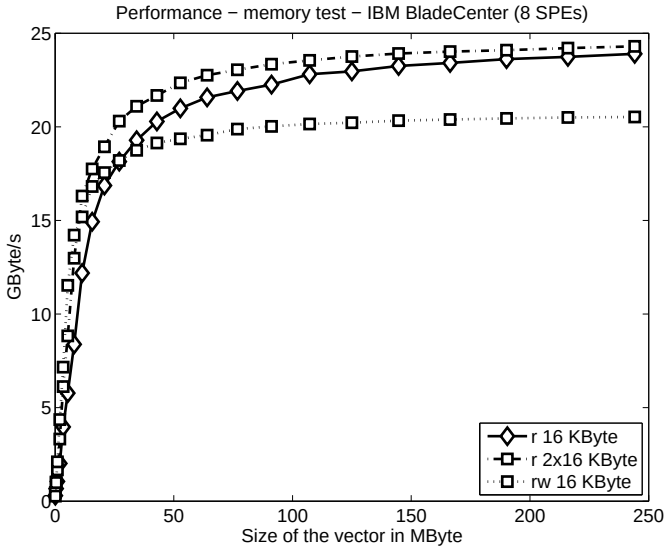


Fig. 2. Main memory bandwidth on QS21, 8 SPEs of one Cell, BIF disabled.

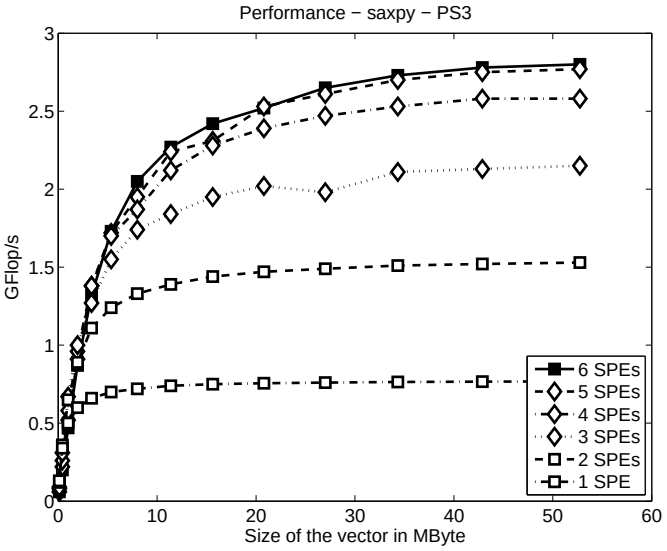


Fig. 3. Performance of `saxpy` vector update on PS3.

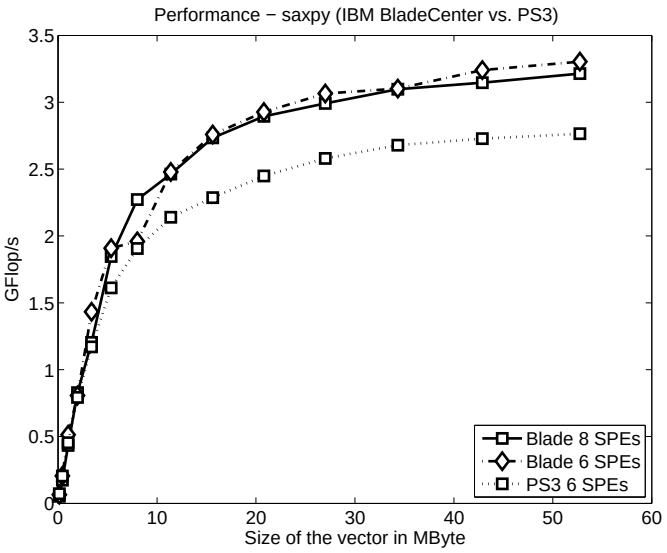


Fig. 4. Performance of `saxpy` vector update on PS3 and QS21.

In the second step we investigate the performance of the RapidMind implementation considering the `saxpy` vector update operation. In- and output arrays for our `saxpy` kernel are one-dimensional arrays (multi-dimensional arrays are not supported in RM on Cell) of 4-tuples related to cubes of size $16 \times 16 \times 4$ (16 KByte). The `saxpy` kernel is a bandwidth-bound operation. Theoretically, there should be no notable dependence on the number of SPEs performing the computation, assuming that each SPE is fed by utilizing maximal available bandwidth. However, measurements of effective bandwidth show that each SPE can be fed with approximately 3 GByte/s only. As depicted in Figure 3, this restriction directly translates to performance of the `saxpy` kernel. For 6 SPEs we only get a maximum performance of 2.8 GFlop/s for vectors of size 52 MByte corresponding to a grid of size 240^3. This performance drop with respect to the deduced upper bound of 4.2 GFlop/s is attributed to the effective bandwidth of about 17.5 GByte/s. Our result is in accordance with the `saxpy` result on Cell in [28] where an effective bandwidth of 17.5 GByte/s is observed as well. Our examination of `saxpy` on QS21 shows a saturation effect. The full available bandwidth (≈ 21.5 GByte/s) is already utilized for five SPEs and there are no performance benefits with six, seven or eight SPEs. In this experiment we disabled communication with the second Cell processor of the blade via the BIF. A performance comparison between PS3 and QS21 shows better results for QS21 in Figure 4. We observe huge performance drops when the vector size is smaller than 20 MByte. On QS21 main memory is 2×1 GByte, hence larger vectors can be treated mitigating the performance drop.

A similar observation applies to the scalar product and the vector norms in Figure 5. We are transferring blocked data of $16 \times 16 \times 4$ 4-tuples and compute local partial sums. Let $N_B = (n/16)^3$ the number of blocks. Both operations perform $2N - 4N_B$ Flop and yield N_B local 4-tuples, which are accumulated by a collective operator. The vector norm kernel requires only a single input array and N memory transfers. Measured bandwidth is 16.5 GByte/s. A slightly better bandwidth can be achieved for the scalar product kernel with 19.1 GByte/s for two input arrays and $2N$ memory transfers. Maximum values for performance without collective operation (labeled with (1) in the legend of Figure 5) are 7.8 GFlop/s for the vector norm and 4.7 GFlop/s, matching the upper bounds with respect to effective bandwidth. With collective operation performance drops to 6.1 GFlop/s (12.5 GByte/s) and 4.0 GFlop/s (16.0 GByte/s).

For the stencil operation, data is blocked into 16 KByte chunks of size $16 \times 16 \times 4$ 4-tuples. The interior nodes within a subcube of size $14 \times 14 \times 2$ 4-tuples are treated first. Due to the smallness of the subcube only $8 \cdot 1568$ Flop are performed per

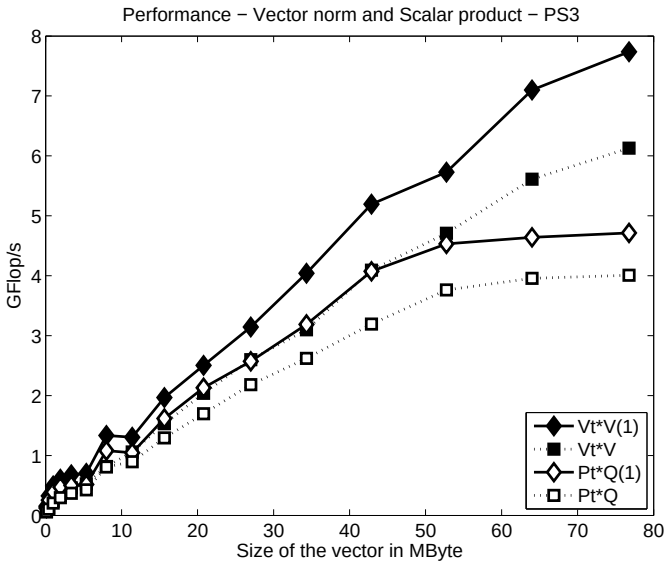


Fig. 5. Performance of vector norms and scalar products on PS3. Label (1) refers to omitted collective operation.

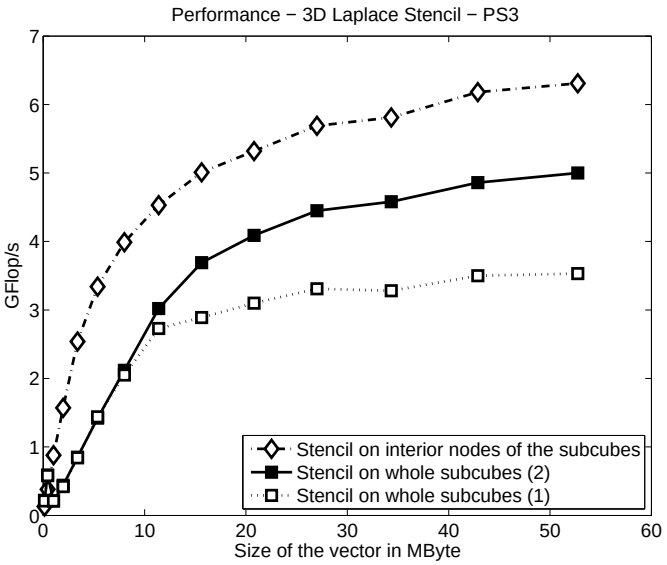


Fig. 6. Performance of stencil operation on PS3.

inner subcube in this step instead of $8 \cdot 4096$ (38%). Effective bandwidth of 17.5 GByte/s gives an upper performance bound of $P_{\text{eff}} \leq 17.5/4 \cdot 1568 \cdot 8/2/4096$ GFlop/s $= 6.65$ GFlop/s. The measured value is 6.4 GFlop/s. Some overhead is attributed to the required swizzle operation with respect to the kernel application to short vectors (Value4f) in z-direction (in x- and y- direction the stencil can be applied to full short vectors without reorganization).

For the treatment of the interfaces between the subcubes, additional data have to be loaded. The upper bound becomes now $P_{\text{eff}} \leq 17.5/4 \cdot 8/2.75$ GFlop/s = 12.7 GFlop/s. This bound is not achieved in the experiment because of additional fetches of small data chunks at the subcube interfaces, the non-regular treatment of several local arrays, accesses to RM-arrays, swizzling, index control, branches due to position of the subcubes, and incorporation of Neumann boundary conditions. Performance results of the stencil operation on PS3 are presented in Figure 6. There, the label (1) refers to the block-lexicographical data layout. Label (2) refers to the results for the data layout where the layers at the interfaces are grouped together.

As a basic observation we find that maximum performance on PS3 only applies to large vectors of size 240^3 (52 MByte) or 272^3 (76 MByte). However, due to main memory limitations the full fluid simulation can run only with vectors of length 112^3 (5 MByte) on PS3. This fact results in a dramatic drop in performance.

Note that the performance results of 21 GFlop/s in SP for the stencil operations on Cell in [25] relate to problems fitting entirely into the LS. Temporal blocking strategies are used in [24] to overcome bandwidth limitations. The authors use layers instead of cubic blocks for spatial blocking. The strategy is to accumulate layers locally and exchange only single layers in each iteration. Temporal blocking techniques do not apply in our situation for two reasons. First, our stencil operation is part of the CG iteration which prevents from executing several steps in one go since intermediate computation of scalar products and vector updates are required. Second, local accumulation of data and temporary exchange of data contradicts the principles of stream processing.

For the advection step (1), we count $f = 108N$ and $w = 12.4N$. Due to the huge number of operations, we find a measured performance of 15.4 GFlop/s on PS3. For the velocity update step (3), we find $f = 9N$ and $w = 10.4N$. Measured performance is 2.6 GFlop/s. Comparably bad performance for both kernels is attributed to data transfers and SIMDization of 3-tuple short vectors (Value3f) instead of optimal 4-tuples and a lot of necessary vector reorganization (swizzling) for the computation.

It is important to note that all computations with our RapidMind version on Cell are restricted to single precision. In practice however, one should mind the trade-off between performance and accuracy with respect to the quality of the simulation results.

VIII. Conclusion

Numerical treatment of large scale 3D flow problems is highly compute time- and memory-demanding and generally necessitates a methodology relying on high performance computing. With the advent of new multi-core technologies, new capabilities become available from the broad market. Prospects of fine-grained parallelism are the key for steady performance increases.

Our investigation of a CFD-solver implementation on Cell shows that RapidMind's stream processing model is providing a simple programming approach in single-threaded manner allowing to take advantage of the parallel capabilities of the Cell BE. The RapidMind Multi-core Development Platform simplifies the development of parallel applications, reducing

the cost and time lines of software development in comparison to multi-threaded projects. Moreover, RapidMind's approach is a step towards portability of implementations on heterogeneous platforms. With respect to the impressive compute capacity of the Cell BE and GPUs, stream processing models like nVIDIA's CUDA or RapidMind offer alternatives with great potential for application areas typically involving structured data. Without necessary insights into hardware peculiarities, the programmer can concentrate on the essentials of the algorithms and does not have to care for the memory transfers and further communication issues. No severe code rearrangements are necessary. Only the compute-intensive parts are assigned to the parallel processing units by invoking stream kernels. The implementation efforts stay moderate and performance results are convincing for kernels with high computational intensity where bandwidth limitations can be hidden. As for well-known hardware technologies, memory organization and data structures remain an issue with huge impact on performance.

A main drawback of the stream processing model is the model-intrinsic absence of concepts to exploit temporal locality and data reuse. Algorithmic flexibility of the numerical methods cannot be expressed in full coverage due to the constraints of the underlying stream processing model. The situation gets worse for problems involving irregular data structures. For bandwidth-bound algorithms - typical algorithms in numerical simulation and treatment of partial differential equations - limitations of the stream programming model in combination with Cell become apparent. Limitations on the Cell BE arise mainly due to the limited main memory bandwidth (although superior compared to many other devices). In our case, main memory limitations lead to short vectors to be transferred resulting in severe bandwidth drops. Limited size of the Local Store of Cell's SPEs is no handicap due to the the restriction on the size of in- and output arrays within RapidMind. In- and output arrays for RM-kernels with size larger than 16 KByte would increase the efficiency of the stencil operation due to the fraction of interior nodes within the subcubes. An essential part of the attained performance of our solver is owed to the memory organization by block-wise lexicographical ordering plus reorganization of the data at the interfaces.

Acknowledgements

The Shared Research Group 16-1 received financial support by the Concept for the Future of Karlsruhe Institute of Technology in the framework of the German Excellence Initiative and the industrial collaboration partner Hewlett-Packard. The QS21 BladeCenter is let by courtesy of SVA System Vertrieb Alexander GmbH, Germany. All mentioned products and brand names are trademarks or registered trademarks of their respective owners.

References

[1] M. Stürmer, J. Götz, G. Richter, A. Dörfler, and U. Rüde, "Fluid flow simulation on the Cell Broadband Engine using the Lattice Boltzmann method," in *ICMMES'07: Proc. 4th Int. Conf. f. Mesoscopic Methods in Engineering and Science*, 2007, accepted.

[2] V. Heuveline and J.-P. Weiß, "Lattice Boltzmann methods on the Clearspeed Advance accelerator board," in *DSFD'07: Proc. 16th Conf. on Discrete Simulation in Fluid Dynamics*, 2007, accepted.

[3] W. B. Langdon and A. P. Harrison, "GP on SPMD parallel graphics hardware for mega Bioinformatics data mining," *Soft Comput.*, vol. 12, no. 12, pp. 1169–1183, 2008.

[4] A. Chorin, "Numerical solution of the Navier-Stokes equations," *Math. Comput.*, vol. 22, pp. 745–762, 1968.

[5] J. Guermond, P. Minev, and J. Shen, "An overview of projection methods for incompressible flows," *Comput. Methods Appl. Mech. Eng.*, vol. 195, no. 44-47, pp. 6011–6045, 2006.

[6] S. McKee, M. Tome, G. Ferreira, J. Cuminato, A. Castelo, F. Sousa, and N. Mangiavacchi, "The MAC method," *Computers and Fluids*, vol. 37, no. 8, pp. 907–930, 2008.

[7] A. Quarteroni and A. Valli, *Numerical approximation of partial differential equations*, 2nd ed. Berlin: Springer, 1997.

[8] P. Gresho and R. Sani, *Incompressible flow and the finite element method. Vol. 1: Advection-diffusion. Vol. 2: Isothermal laminar flow.* Chichester: Wiley, 2000.

[9] D. Lukarski, "Specific aspects of a parallel implementation of a 3D CFD solver on the Cell architecture," Master's thesis, Dept. Math., Univ. Karlsruhe, Germany, 2008.

[10] Y. Saad, *Iterative methods for sparse linear systems*, 2nd ed. Philadelphia: SIAM, 2003.

[11] *Cell Broadband Engine architecture*, Ver. 1.02, IBM, 2007.

[12] A. Arevalo, R. Matinata, M. Pandian, E. Peri, K. Ruby, F. Thomas, and C. Almond, *Programming the Cell Broadband Engine, Examples and Best Practices*, IBM Redbooks, 2008.

[13] *CorePy: Synthetic Programming in Python*. Indiana Univ., http://www.corepy.org/, 2008.

[14] M. Ohara, H. Inoue, Y. Sohda, H. Komatsu, and T. Nakatani, "MPI Microtask for programming the Cell Broadband Engine processor," *IBM Syst. J.*, vol. 45, no. 1, pp. 85–102, 2006.

[15] *Sequoia*. Stanford Univ., http://sequoia.stanford.edu/, 2008.

[16] *Compiler Technology for Scalable Architectures*. IBM Research, http://www.research.ibm.com/cellcompiler/, 2008.

[17] P. Bellens, J. M. Perez, R. M. Badia, and J. Labarta, "CellSs: a programming model for the Cell BE architecture," in *SC'06: Proc. 2006 ACM/IEEE Conf. on Supercomputing*. New York: ACM, 2006, p. 86.

[18] B. Bouzas, R. Cooper, J. Greene, M. Pepe, and M. Prelle, 2000, Mercury Computer Systems, http://www.mc.com/uploadedFiles/MCF-API-Conf-Paper.pdf.

[19] M. D. McCool, "Scalable programming models for massively multicore processors," *Proc. IEEE*, vol. 96, no. 5, pp. 816–831, 2008.

[20] *RapidMind Multi-Core Software Platform - User Guide*. RapidMind, 2007.

[21] *RapidMind Multi-Core Software Platform - API Reference Manual*. RapidMind, 2007.

[22] *RapidMind Developer Portal*. RapidMind, https://developer.rapidmind.net, 2008.

[23] *Compute Unified Device Architecture (CUDA)*. nVIDIA, http://nvidia.com/cuda, 2008.

[24] K. Datta, S. Kamil, S. Williams, L. Oliker, J. Shalf, and K. Yelick, "Optimization and performance modeling of stencil computations on modern microprocessors," *SIAM Review*, to appear, 2008.

[25] S. Williams, J. Shalf, L. Oliker, S. Kamil, P. Husbands, and K. Yelick, "Scientific Computing Kernels on the Cell Processor," *Int. J. Parallel Program.*, vol. 35, no. 3, pp. 263–298, 2007.

[26] S. Kamil, K. Datta, S. Williams, L. Oliker, J. Shalf, and K. Yelick, "Implicit and explicit optimizations for stencil computations," in *MSPC'06: Proc. 2006 workshop on Memory System Performance and Correctness*. New York: ACM, 2006, pp. 51–60.

[27] G. H. Golub and C. F. Van Loan, *Matrix computations*, 3rd ed. Baltimore: Johns Hopkins Univ. Pr., 1996.

[28] T. J. Knight, J. Y. Park, M. Ren, M. Houston, M. Erez, K. Fatahalian, A. Aiken, W. J. Dally, and P. Hanrahan, "Compilation for explicitly managed memory hierarchies," in *PPoPP'07: Proc. 12th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*. New York: ACM, 2007, pp. 226–236.

OPTIMISING COMPONENT COMPOSITION USING INDEXED DEPENDENCE METADATA

Lee W. Howes, Anton Lokhmotov, Paul H. J. Kelly, A. J. Field

Department of Computing, Imperial College London
email: {lwh01, anton, phjk, ajf}@doc.ic.ac.uk

ABSTRACT

This paper explores the use of *dependence metadata* for optimising composition in component-based parallel programs. The idea is for each component to carry additional information about how points in its iteration space map to memory locations associated with its input and output data structures. When two components are composed this information can be used to implement optimisations that would otherwise require expensive analysis of the components' code at the time of composition. This dependence metadata facilitates a number of cross-component optimisations – in this paper we focus on loop fusion and array contraction. We describe a prototype framework, based on the CLooG loop generator tool, that embodies these ideas and report experimental performance results for three non-trivial parallel benchmarks. Our results show execution time reductions of up to 50% using the proposed framework on an 8 core xeon.

1. INTRODUCTION

Component based programming consists of writing software entities to fulfill specified interfaces. Component models allow multiple component implementations to satisfy the same interface, offering flexibility on the choice of implementation for a particular problem or computing platform. However, treating components as black boxes described by their interfaces can limit the scope for optimisation. In particular, whilst individual components can be statically optimised when the component is defined, component compositions can only be optimised at the point of use. This requires an element of dynamic optimisation that exploits context information.

Powerful but expensive inter-procedural compiler optimisations such as enabled by the polyhedral framework [1] could be used once the composite component structure is known. However, the cost of the analysis would have to be paid each time the same components were composed in the same way.

Adaptive components are explicitly programmed to make use of context information, e.g. knowledge of the components with which they are composed, in order to produce optimised execution schedules. In this paper we propose to implement a form of adaptive behaviour through the use of supplied component metadata and to use that metadata to identify dynamic optimisation opportunities at the time of composition. The fact that the metadata is supplied rather than extracted at composition time, obviates the need to analyse a component's code each time it is used, in order to identify whether cross-component optimisation opportunities exist.

The metadata we explore in this paper, which we refer to as *indexed dependence metadata*, defines the set of memory locations that a component may access at a particular point in its iteration space. The relationship between these mappings in different components serves to define implicitly the communication requirements of their compositions.

By examining the memory dependence metadata of the components in a composition, we seek to expose opportunities for cross-component optimisation that are not possible by optimising the individual components in isolation.

Specifically, in this paper we use the dependence metadata to determine whether two loops occurring separately in the components of a composition can be aligned whilst respecting dependences, in which case the loops can be fused. Fusion in turn may facilitate array contraction, reducing the space requirements of the composition, and inter-processor communication in the case where the components themselves comprise parallel loops. We use CLooG [2, 3] to generate the code for a fused loop using a scheduling matrix generated from an analysis of the components' metadata and a matrix representation of the iteration space generated from the components' source code.

The contributions of the paper are as follows:

- We introduce the idea of indexed dependence metadata, which defines the set of memory locations that may be read from and written to by a component at each point in its iteration space (Section 3).

- We show how the dependence metadata can be used in conjunction with a representation of the components' iteration spaces to implement loop fusion and array contraction across the component boundaries in a composition (Section 5). In particular, we extend this to parallel components, where the contraction reduces inter-processor communication.

- We describe a prototype software component framework incorporating the above ideas, which has potential applications in multi-core software development (Sections 2 and 4).
- We illustrate the power of the approach by showing substantial performance improvements through fusion of parallel components in linear algebra and image processing benchmarks and a 3D multigrid solver (Section 6). On an eight-core Intel Xeon system, maximum performance improvements on these examples range from 12% to 50%.

```
<interface id="iContourfilter">
    <input type="float" name="image_in"
        format="array(in_x,in_y)" />
    <output type="float" name="image_out"
        format="array(out_x,out_y)" />
</interface>
<interface id="iConvolution">
    <input type="float" name="image_in"
        format="array(in_x,in_y)" />
    <input type="float" name="filter_in"
        format="array(filter_x,filter_y)" />
    <output type="float" name="image_out"
        format="array(out_x,out_y)" />
</interface>
```

Listing 1. Interface specifications for the contour filter and convolution.

```
<component id="cf" >
  <implements id="iContourfilter" />
  <uses name="conv">
    iConvolution(
      image_in(in_x, in_y), filter_in(3, 3),
      image_out(out_x, out_y) flow to F1)
  </uses>
  <constraint type="equality">
    conv.in_x=in_x
  </constraint>
  ...
</component>
```

Listing 2. Part of the contourfilter component specification.

2. ARCHITECTURE OVERVIEW

Our component programming system is designed to select and generate code from a library of components. Components carry metadata describing functional interfaces and data dependence relationships. We identify three elements: *Component*, *Interface* and *Manager*.

The application and individual *components* depend on one or more *interfaces*. Components also implement interfaces, satisfying the contract defined by the interface. The *manager* maintains the component dependence graph and allocates component implementations to the interfaces as necessary. If a component $C1$ depends on an interface that is implemented by a component $C2$, we say that $C2$ is a subcomponent of $C1$. We generate the dependency graph of an application by recursively expanding the dependencies in the component graph. The assignment of components to interfaces is performed during a later graph pass.

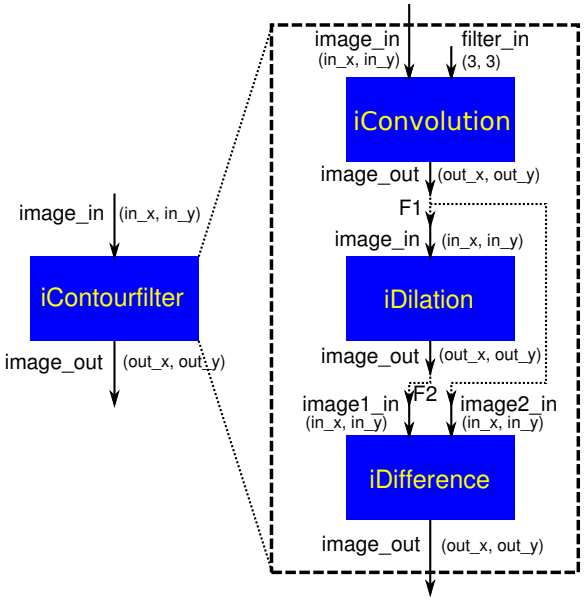


Fig. 1. A contour filter example showing dependencies, data flows and size descriptions of inputs and outputs.

Figure 1 shows the dependency relationships for an image filtering example. We see a *iContourfilter* interface with one input and one output, implemented by a component that depends on *iConvolution*, *iDilation* and *iDifference* interfaces to perform its computation. Flow annotations *F1* and *F2* define data flow dependencies at the composition level.

Listing 1 shows the specification for two of the interfaces in Figure 1: *iContourfilter* and *iConvolution*. Listing 2 shows part of the component specification for the contour filter (*cf*), including its dependence on its convolution subcomponent.[1] The *cf* component, which implements the *iContourfilter* interface, depends on the *iConvolution* interface. We name the dimensions of the input and output parameters, and specify a constant 3×3 size for the filter parameter. The *flow to* keyword names a data flow as in Figure 1.

The implementation language for a given component is flexible. We currently support C/C++, a high level polyhedral representation of C, or pre-compiled binaries. In principle the system can integrate components in any language, given support in the component manager.

[1]Note that our implementation currently uses XML to define interfaces, component specifications and dependence metadata, although we envisage the use of automated or GUI based tools in the future.

3. COMPONENT METADATA

In general, the input and output variables of components need to interact with those in their subcomponents. For example, variables in subcomponents can be configured to share values of variables in the parent component, and hence values can propagate through the component graph. Additional metadata can be attached to a component specification in order to express these properties. For example, Listing 2 shows an equality constraint specifying that the value *in_x* in the interface matches the *in_x* in the subcomponent named *conv*.[2] Additionally, data can flow from one subcomponent to another, and hence through various levels of the component graph when combined with parent/child relationships. In the example, the *image_out* value of *iConvolution* is connected (*flows to*) the flow *F1*, which will be connected again to an input variable in another dependency of the component. Component graph data flows are defined in the metadata, to avoid composition-time component analysis.

It should be emphasised that the aim is to provide dependence relationships on the component inputs and outputs *at composition time*, without analysis of the component code; indeed, this code might be in binary form, which could preclude such analysis.

3.1. Indexed Dependence Metadata

Indexed dependence metadata defines a set of memory addresses that a component may access at a point in its iteration space. By interpreting the metadata, the component manager can map a given set of iterations onto a set of memory locations and, assuming predictable and reasonably simple patterns, can infer dependencies across sets of iterations.

In Figure 2 we see the region constraints of our convolution filter from the running example, assuming a 3×3 filter. Listing 3 shows the generic component specification for the convolution filter assuming an arbitrary-sized filter. The specification includes various pieces of metadata that the component manager can use to optimise the composition to its context. Note that omitting some or all of the metadata will not break the code; it will simply limit the scope for optimisation.

The iteration space of the component corresponds to the indices into the input image (*image_in*), as shown. For each point in the iteration space a 3×3 rectangular region of *image_in*, relative to the point, will be read. This corresponds to a *radius* of size 1 in each dimension around the point. Additionally, the whole of *filter_in* will be read and the corresponding point in *image_out* (i.e. a radius of size 0 in each dimension) will be written. The filter input variables are defined in the interface and their values propagated through the component graph.

[2] To generalise this, we can specify *inequalities* rather than *equalities* to constraints, and hence define the possible ranges for subcomponent parameters. Relaxing the requirements of a subcomponent can allow more specific and efficient subcomponents to be selected.

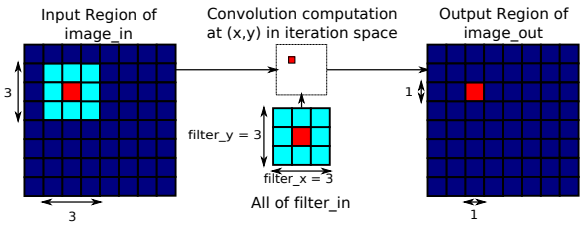


Fig. 2. Region dependencies at a point in the iteration space.

```
<component id="convolution">
  <iteration_space
    dimensions="(image_in.width,image_in.height)"
  />
  <constraint type="dependentregion"
    shape="rectangle">
    <constraintinput name="image_in"
      placement="relative"
      radius="((filter_in.w-1)/2,(filter_in.h-1)/2)"
    />
    <constraintinput name="filter_in"
      placement="absolute"
      range="(0->filter_in.w-1,0->filter_in.h-1)"
    />
    <constraintoutput name="image_out"
      radius="(0,0)" />
  </constraint>
</component>
```

Listing 3. Constraints in the specification of a component.

3.2. Component relationships through metadata

Metadata directly affects the relationships between components. If two components communicate either through a functional dependence, or through a data flow, the metadata will need to be propagated.

A component's metadata must be combined with the metadata of other components to give a full specification of a relationship. For example, in Listing 2 the contour filter requires a 3×3 convolution operation, which defines an access region on its input. The size of this access region depends on the size of the filter. Therefore, to specify fully the convolution's metadata we need to propagate the filter size specified by the contour filter through the graph. This propagation can be achieved by passing metadata bindings through parent/child and data-flow relationships.

When the application requests an interface, values are bound to the interface's parameters. These values are combined with constraints and dependence metadata throughout the component graph to bind values to variables and define component relationships as accurately as possible. Component selection or composition uses the propagated information to limit the binding of components to interfaces or to

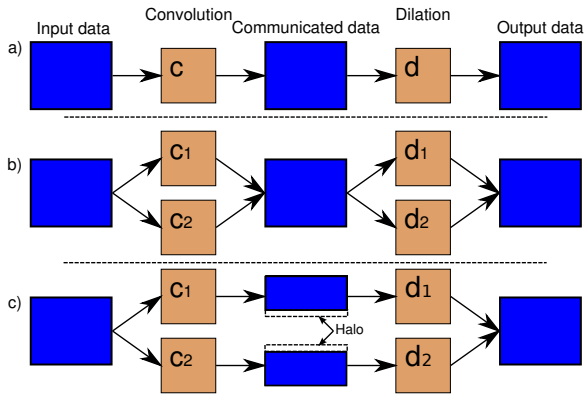


Fig. 3. The addition of region descriptors enables more efficient parallelism.

define possible composition optimisation opportunities.

Figure 3 shows how the information provided by combining region definitions with the size of the dataset can reduce the size of the required communication between two components, in this case the convolution and dilation components from Figure 1. Figure 3(a) is an example of a simple component composition communicating via an intermediate data set. If we parallelise the components with no knowledge of the components' internals, we do not know how much of the data each thread will need and must communicate it all. In this case the individual components would be parallel but not their composition, as illustrated in Figure 3(b).

With full region information we can minimise the communication between parallel components. For example, if the dilation component depends also on a 3×3 filter then parallelisation of the component as shown in Figure 3(c) requires only half the data set, plus an additional halo strip, to be sent from each convolution thread to its corresponding dilation thread. As a consequence, data can be kept in more localised, faster, memory for longer and communication is more predictable. If c_i and d_i both execute in the same memory region, only the halo strips would need to pass through higher levels in the memory hierarchy.

3.3. Scalability

The component metadata in the examples are currently written by hand. We envisage that in practice the information will be, at least partially, obtained by component analysis at construction time. Clearly, complicated components limit the feasibility of analysis. By limiting the dependence information to the input and output data structures of the component, and assuming the contents are correct, we simplify the run time workload, and improve scalability in that manner, ensuring that the complexity of individual components does not affect composition time scalability. Generation time analysis may not be possible for all components. However, the discussed system localises analysis at construction time and, as a result, increases the possibility of correct dependence construction over fully general system-wide analysis of all possible interactions.

4. CODE GENERATION

Our system supports components in various forms. In the simplest case we use a pre-compiled binary, which is linked at run time. Alternatively, we can compile and link a component code at run time. Delaying compilation to run-time offers scope for performance improvements as the compiler may have more information about the code, or the system.

A further possibility is to generate code at run time, before compilation and linking. Earlier work such as Taskgraph [4] shows that run time code generation and compilation can be effective. In this system we view both run time code generation and compilation as a lowering from one implementation level to another. For example, we can lower from a high level source representation, to C++; then through compilation of C++ to a binary. Each stage takes a component as input, and generates a replacement component as output, with correct lowered annotations. This approach is flexible and conveniently supports component caching.

We use the CLooG [2, 3] code generator to construct the code for compilation. CLooG-based components are high-level representations of iteration spaces, and are converted to C++ components in the first stage of the lowering process.

```
COMPONENT_TARGET(difference)
{
  POLYHEDRAL_LOOP(i) [ i < image1_in.height();
      i >= 0; ] {
    POLYHEDRAL_LOOP(j) [ j < image1_in.width();
        j >= 0; ] {
      image_out(x,y) = image1_in(x,y)-image2_in(x,y)
    }
  }
}
```

Listing 4. A simple polyhedral representation of the iteration space of an image difference operation.

CLooG is based on the polyhedral model [1] which represents execution schedules as polyhedra in multi-dimensional iteration spaces. CLooG's input defines a polyhedral iteration space using a set of affine half-spaces as individual inequalities in the rows of a matrix. An example of the input matrix can be seen in Figure 5(b). CLooG outputs the code necessary for each statement to visit each integer point within the polyhedron. CLooG does not perform dependence analysis and so for ill-considered input will generate incorrect output. As a result, our input to the code generator must satisfy data dependencies.

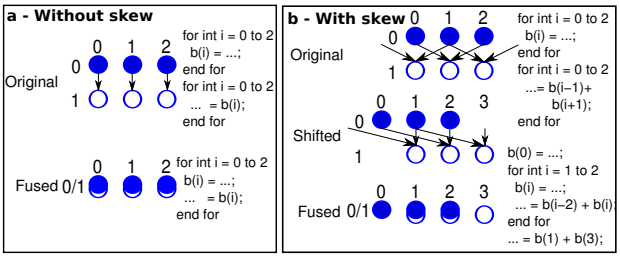


Fig. 4. A simplified one-dimensional loop fusion example.

We generate input to CLooG from a component implementation as in Listing 4. Full analysis of C code or a binary representation of analysed dependencies as polyhedra would work equally well but this syntax offers us a simple basis to work with for experimentation. We specify the execution polyhedron of the kernel using nesting to define dimensions and lists of inequalities to define ranges for the variables. This inequality syntax is converted into CLooG's input matrices during the process of lowering from CLooG input to C++. CLooG is capable of generating hundreds or thousands of lines of code to cover complicated iteration spaces which would be extremely difficult to write by hand.

5. USING METADATA FOR OPTIMISATIONS

The presence of dependence metadata on components allows the *manager* to perform component mapping decisions and, in addition, cross-component optimisations. In this work we illustrate the potential by applying loop fusion (and the enabled array contraction) to a connected subgraph of components.

5.1. Increasing temporal locality with loop fusion

Loop fusion [5] takes two or more consecutive loops and merges the bodies together as illustrated in Figure 4(a). Fusion reduces the number of control instructions, improves the temporal locality of data and, when fusing parallel loops, avoids unnecessary synchronisation (albeit with the risk of harming cache performance or instruction scheduling).

Loop dependencies can complicate fusion. In Figure 4(b) for example, statement 1 has a forward data dependence on the output of statement 0. These two statements from the same iteration number of the original loops cannot execute in the same iteration of the fused loop. The dependence can be resolved by shifting the iteration space of the second loop. The shift allows each loop to perform its given set of iterations with all dependencies satisfied before the data is required. The result of this fusion and shift (sometimes called "shift and peel" [6]) is a guarded or partially unrolled loop nest as in Figure 4(b), with a necessary loss of parallelism at the edges.

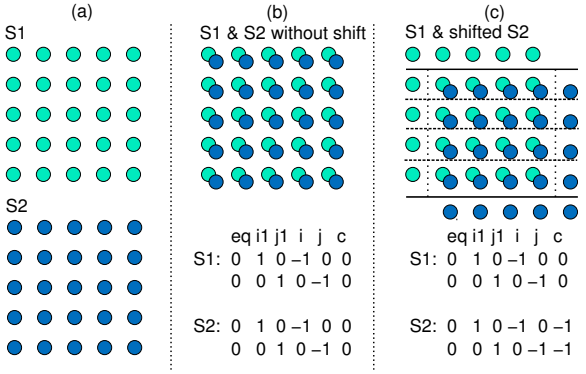


Fig. 5. The scatter matrix can be used to schedule the loop by changing the logical execution time of a given iteration.

Input and output regions defined in the metadata make the data dependencies explicit. We know which data values may be read or written at a given point in a component's iteration space, and hence can compute the shift necessary to resolve data dependencies.

We use CLooG to generate code representing the fused set of components. We supply the individual input matrices that define the iteration space. We also provide a mapping of points in the iteration space to a logical execution time, known as the *scatter* matrix. As demonstrated in Figure 5, we can specify that a point (i, j) in the iteration space (a) of a component can be mapped to (t_i, t_j) in time, where either $t_i = i$ and $t_j = j$ (b), or $t_i = i + 1$ and $t_j = j + 1$ (c), shifting the schedule.

The amount of shift required depends on the dependence relationship between two components. These relationships are computed from the access region metadata. For example, a 3×3 region as input to the second component requires a shift of 1 in the iteration space of the second component so that the output of the first is ready when it is needed. In the general case, we need to compute the last iteration in the source component that may generate data needed by the matching iteration in the target component. If the dependence distance is constant, we can compute a static schedule correction. We parameterise the scatter matrix by a set of shift values computed from the dependence relationships to shift the logical time of the component and therefore of its statements. With a correct scheduling defined in the scatter matrix, CLooG will generate a series of loops that respects the inter-component data dependencies.

Component selection for fusion depends on the flow of data between components. Unrelated components are easy to fuse, but unlikely to benefit from fusion. Components that share inputs, or communicate using a intermediate data structure, are more likely to benefit. Having analysed the data flow in the parent component at construction time, we can fuse the children at composition time. Calls to the sub-

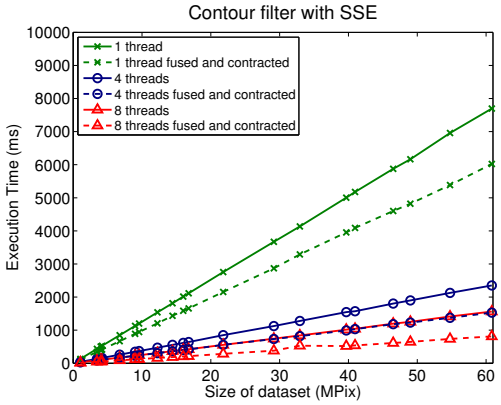


Fig. 6. Execution time of the contour filter example.

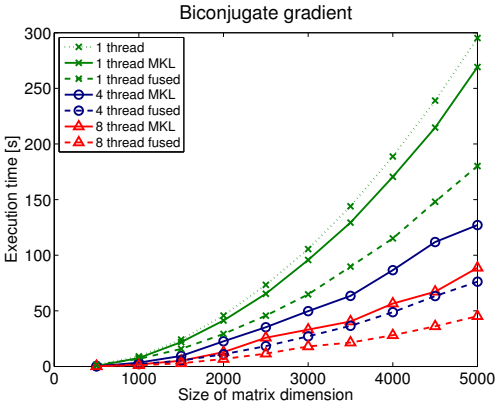


Fig. 7. Comparing MKL, with custom version of biconjugate gradient for 1, 4 and 8 threads (custom without fusion shown only for 1 thread).

components can be replaced with calls to stub functions that merely prepare data structures. The execution of the fused component can be delayed until the last subcomponent call. As a result, the parent component itself need not change.

5.2. Reducing storage through contraction

Loop fusion reduces the period between generation and use of intermediate data values, often leading to more efficient use of the cache and improved performance. Array contraction offers further scope for improvements and can be a key enabler of high performance in large parallel fused loops [7]. Rather than storing entire intermediate arrays, we reduce the intermediate storage to the minimum required to satisfy data flow requirements, reducing the use of memory bandwidth due to cache displacement.

6. EXPERIMENTAL RESULTS

We implement three examples using our component framework to demonstrate its capabilities and how we can improve the performance of an application. These examples possess different data flow situations and hence show varied performance.

To enable fusion, all subcomponents are implemented in a high-level polyhedral representation, as in Listing 4, and have appropriate dependent region and data flow metadata attached to describe the relationships between component inputs and outputs.

We compile using Intel C/C++ 10.1 or GCC 4.2 (whichever performs better) on an eight core, dual-socket Intel Xeon based machine running a 64-bit Linux 2.6 kernel and parallelise using OpenMP. The single threaded code is the unparallelised, sequential version.

6.1. Image processing

The contour filter (Figure 1) operates on four-component (RGBA) data and is vectorised using SSE instructions.

The dilation subcomponent selects a maximum value in a region of the output of the convolution subcomponent. To allow for this dependence, the fused execution space must shift. The execution of the elements of both the dilation and difference are delayed by the radius of the region.

Figure 6 shows performance results for the contour filter with SSE. There is a substantial reduction in execution time for fusion combined with contraction. Execution time is reduced by 21% for a single thread, 35% for four threads and 48% for eight threads. While not plotted on the graphs, fusion alone offers 4%, 11% and 20% respectively. The improvement from fusion alone is slightly erratic, but tends to decrease with data set size as the larger range of visited addresses increases the chance of an individual element being removed from the cache. A similar effect is not seen with the contracted data sets where the accessed address range is reduced to a circular buffer of a few image rows in size.

6.2. Linear Algebra

Our linear algebra example is a biconjugate gradient solver from the Iterative Template Library [8], with components defining various aspects of the computation flow. We allow fusion to occur between a standard matrix/vector multiplication, and a transposed matrix/vector multiplication. Note that in this benchmark we share input matrices between components, rather than having data flow from one component to another. A result of this lack of data flow is that there is no communicated array to contract and hence this example supports fusion only.

In this example we use 1×1 access regions because the execution maps a single iteration space point to a single data

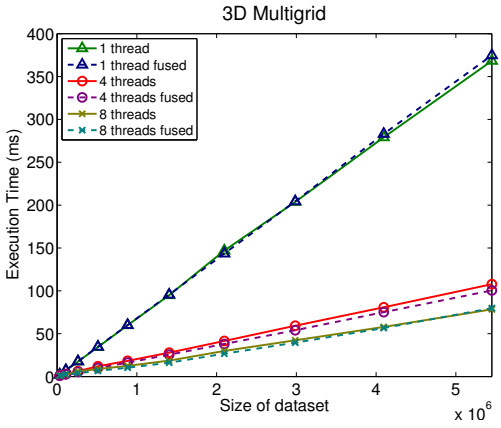


Fig. 8. Execution time for single and four and eight threaded 3D multigrid solver kernel.

element from each input. As the input and output vectors are present in a single dimension only, the mapping is a projection onto that dimension.

Figure 7 graphs performance results for the fused versions of the biconjugate gradient solver as well as results for Intel's Math Kernel Library (MKL) [9] as a baseline with a comparison with the original version on a single thread. We can see that while there is an improvement in performance over MKL for all numbers of threads, this improvement is more pronounced for 4 and 8 threads where memory contention between cores is reduced by fusion.

6.3. 3D Multigrid

Multigrid solves differential equations using a hierarchy of discretisation levels. We adapted this example from the NAS Parallel benchmarks suite [10] using fixed boundary conditions.[3] We created a sequence of dependent components based on the core functions that iterate on the data: *Data initialisation*, *Interpolation from a lower resolution computation stage*, *Computation of residuals* and *Application of a smoother to the data*.

The four components are related by region and data flow dependencies describing how a value in the iteration space of one component relates to a value in the iteration space of the next in sequence. We require $3 \times 3 \times 3$ regions around the input to the interpolation, residual and smoother applications. We make the kernel more efficient by absorbing the inner dimension of the loop nest, allowing hand tuning of the inner loop. Given such a kernel, our access region specifies an entire row of the data set in one dimension and a 3×3 region in the other two. Note that the component manager need not know that our tuned kernel has a carefully written inner loop, only that it needs to access an entire row of the data set to perform its work.

Figure 8 offers performance results for 1, 4 and 8 threads. The improvement from fusion peaks at 4 threads where we see a mean reduction in execution time of 12% over the range shown. For larger data sets the performance of fusion falls off as the amount of data maintained by the 3D computation skew creates stress on the cache and other shared data structures of the CPU. The peak at 4 threads is similarly explained because the L2 cache is shared between pairs of cores, reducing the effective cache size per core when 8 threads are used.

[3] In the original code the computation is complicated by a cyclic dependency due to a wrap-around boundary condition. While fusion is still possible with the cyclic dependency, performance benefits are lost due to the increased loop shift necessary to support wrapping on all three dimensions.

7. RELATED WORK

Adaptive component models have been widely studied, for example in embedded systems (e.g. see [11]), as well as more generally in distributed systems (e.g. [12]). Dowling and Cahill [13] offer a useful framework, emphasising the importance of separating adaptational from computational code. Recent work on the Common Component Architecture (CCA) looks at composing, substituting and reconfiguring components during application execution [14].

Our component composition builds on work on Architecture Description Languages (ADLs) such as Darwin [15] and xADL [16], and is similar to Think [17]. Our work differs from other ADLs in its support for indexed dependence metadata, that denotes dependence relationships for individual iteration space points.

CLooG arises from Bastoul's work [3] and builds on earlier work on code generation in the polyhedral model by Griebl and Wetzel [1]. Griebl [18] applies the polyhedral model to parallelisation of loop nests while recent work by Pop et al. [19] looks at integrating polyhedron based analysis into GCC.

This paper is an attempt to realise the THEMIS [20] proposal and is part of a larger body of work including the Task-graph [4] library, related work from Cornwall [7] and active libraries in linear algebra from Russell [21].

ZPL [22] (a precursor of Chapel [23]) and KeLP [24] (which led to Chombo [25]) had explicit regions - in fact a "region calculus". However their regions represent partitions of iteration and data spaces - whereas in this work we represent the mapping between points in the iteration space and memory locations.

Languages like StreamIt [26] use the concept of sequences of data items, called *streams*, which are operated on by pure functions, called *filters*. Clear (and often static) data-flow relationships between filters enable cross-component optimisations. In contrast, our framework enables cross-component optimisations for general programs operating on arbitrary data sets.

8. CONCLUSIONS AND FUTURE WORK

We have shown how interfaces with indexed dependence metadata can be used to improve the performance of component compositions. Our experimental results show that metadata can be used to perform aggressive component fusion, generating hundreds of lines of code (200-300 in the contour filter and over 1500 for the multigrid example) that would be challenging to implement by hand. We have also confirmed that loop fusion can substantially reduce execution time through improvements in temporal locality of data.

The THEMIS proposal discusses more possibilities for metadata than we have been able to implement to date. In the future we hope to proceed further with this investigation, particularly in the area of applying cross-component optimisation techniques to data layout by adding metadata annotations describing the access patterns for data. More varied access descriptors and tighter integration into the programming language using C++ pragmas or compiler support for iterator classes are other targets.

The multigrid example shows that in some cases fusion gives only a small benefit. In these cases we plan to use adaptive component mapping to use the original components rather than fused sets when a fusion attempt reduces performance. Optimal combinations may include calls to vendor libraries wrapped in components, as used in the MKL comparison for the linear algebra example.

Novel architectures such as heterogeneous multicore platforms require novel optimisation strategies. Hand coding is often impractical. We envisage that adaptive, metadata-driven optimisation techniques will be of increasing relevance as technology develops.

9. REFERENCES

[1] M. Griebl, C. Lengauer, and S. Wetzel, *Code generation in the polytope model*, Proc. PACT, IEEE Comp. Soc., 1998.

[2] *CLooG*, http://www.cloog.org/.

[3] C. Bastoul, *Code generation in the polyhedral model is easier than you think*, Proc. PACT, IEEE Comp. Soc., 2004.

[4] O. Beckmann, A. Houghton, P. H. J. Kelly, and M. Mellor, *Run-time code generation in C++ as a foundation for domain-specific optimisation*, Proc. Domain-Specific Program Generation International Seminar, 2003.

[5] K. Kennedy and K. S. McKinley, *Maximizing loop parallelism and improving data locality via loop fusion and distribution*, Proc. LCPC, Springer, 1994.

[6] N. Manjikian and T. S. Abdelrahman, *Fusion of loops for parallelism and locality*, IEEE Trans. Parallel Distrib. Sys.

[7] J. L. T. Cornwall, P. H. J. Kelly, P. Parsonage, and B. Nicoletti, *Explicit dependence metadata in an active visual effects library*, Proc. LCPC, Springer, 2007.

[8] A. Lumsdaine, L.-Q. Lee, and J. Siek. *Iterative template library*, http://www.osl.iu.edu/research/itl/, 2001.

[9] Intel. *Math Kernel Library*, 2008

[10] B. L. Chamberlain, S. J. Deitz, and L. Snyder, *A comparative study of the NAS MG benchmark across parallel languages and architectures*, Proc. SC, IEEE Comp. Soc., 2000.

[11] H. Ma, I.-L. Yen, F. Bastani, and K. Cooper, *Composition analysis of QoS properties for adaptive integration of embedded software components*, Proc. ISSRE, 2003.

[12] L. Baresi, S. Guinea, and G. Tamburrelli, *Towards decentralized self-adaptive component-based systems*, Proc. SEAMS, ACM, 2008.

[13] J. Dowling and V. Cahill, *The k-component architecture meta-model for self-adaptive software*, Proc. of the Third International Conference on Metalevel Architectures and Separation of Crosscutting Concerns, Springer, 2001.

[14] L. C. McInnes et al, *Computational quality of service for scientific CCA applications: Composition, substitution, and reconfiguration*, Argonne Nat. Lab., Tech. Rep. 2006

[15] J. Magee, N. Dulay, S. Eisenbach, and J. Kramer, *Specifying distributed software architectures*, Proc. European Software Engineering Conference, Springer, 1995.

[16] E. Dashofy, A. van der Hoek, and R. Taylor, *A highly-extensible, XML-based architecture description language*, Software Architecture, 2001.

[17] A. E. Özcan, O. Layaida, and J.-B. Stefani, *A component-based approach for MPSoC SW design: Experience with OS customization for H.264 decoding*, ESTImedia, IEEE Comp. Soc., 2005.

[18] M. Griebl, *Automatic Parallelization of Loop Programs for Distributed Memory Architectures*, Habilitation Thesis, University of Passau, 2004

[19] S. Pop, G.-A. Silber, A. Cohen, C. Bastoul, S. Girbal, and N. Vasilache, *GRAPHITE: Polyhedral analyses and optimizations for GCC*, Proc. GCC Summit, 2006.

[20] P. Kelly, O. Beckmann, A. J. Field, and S. Baden, *THEMIS: Component dependence metadata in adaptive parallel computations*, Parallel Processing Letters, 2001

[21] F. P. Russell, M. R. Mellor, P. H. J. Kelly, and O. Beckmann, *An active linear algebra library using delayed evaluation and runtime code generation*, Proc. LCSD, 2006.

[22] B. L. Chamberlain, E. C. Lewis, C. Lin, and L. Snyder, *Regions: an abstraction for expressing array computation*, SIGAPL APL Quote Quad, 1998.

[23] B. Chamberlain, D. Callahan, and H. Zima, *Parallel programmability and the chapel language*, Int. J. High Perf. Comp. Appl., 1997.

[24] S. J. Fink, S. B. Baden, and S. R. Kohn, *Efficient run-time support for irregular block-structured applications*, J. Parallel Distrib. Comp., 1998

[25] P. Colella et al. *Performance and scaling of locally-structured grid methods for partial differential equations*, SciDAC 2007 Annual Meeting.

[26] W. Thies, M. Karczmarek, and S. Amarasinghe, *StreamIt: A Language for Streaming Applications*, Proc. Compiler Construction, Springer, 2002.

Accelerating Stencil-Based Computations by Increased Temporal Locality on Modern Multi- and Many-Core Architectures*

Matthias Christen[1] m.christen@unibas.ch
Olaf Schenk[1] olaf.schenk@unibas.ch
Peter Messmer[2] messmer@txcorp.com
Esra Neufeld[3] neufeld@itis.ethz.ch
Helmar Burkhart[1] helmar.burkhart@unibas.ch

***Abstract*—Stencil computations arise in a wide range of applications of computational sciences. This paper focuses on stencil computations arising in the context of a biomedical simulation. Compute-intensive bio-medical simulations represent an attractive application for the Cell Broadband Engine Architecture (CBEA) and for graphics processing units (GPUs) as hardware accelerators. Due to the low arithmetic intensity of stencil computations and bandwidth limitations of the compute hardware, the performance is usually only a fraction of peak performance. We detail an implementation of parallel stencil computations on the CBEA and GPUs, which improves performance by exploiting temporal locality. We report on performance improvements over CPU implementations.**

***Index Terms*—Stencil computation, Cell processor, GPGPU, parallel programming, performance measurement, biomedical simulation**

I. Introduction

Stencil computations arise in a wide range of applications of computational sciences. A main application of stencil-based computations are numerical PDE solvers that use a finite difference or multigrid method [7], where stencil computations are typically performed in the smoothing step. Image processing is another field in which stencils play a major role. In stencil-based computations, each node in a multi-dimensional grid is updated with weighted values contributed by neighboring nodes.

Novel microarchitectures such as the the Cell Broadband Engine Architecture and alternative hardware, such as GPUs, have become of interest to the scientific computing community. While commodity GPUs are available at very low prices, they nevertheless deliver amazing computational power. Both the Cell BE and GPUs outperform commodity desktop CPUs by an order of magnitude. Both architectures are inherently parallel.

In this paper a specific stencil type is considered, arising in a biomedical simulation for hyperthermia cancer treatment. The simulation is ported to the Cell BE, a GPU system, and a CPU cluster. On the cluster MPI is used for parallelization. In an ongoing work, the code is also used to parallelize for a Cell BE cluster. Algorithms are discussed that are used to boost performance.

In [5] and [17], optimizations for stencil computations are discussed and benchmarks of the code running on state-of-the-art microprocessors including the Cell BE processor are presented. In this work, in contrast to [5] and [17], a more versatile stencil stemming from a real-world application is considered, which entails additional challenges. In particular, a lot more data has to be transferred per calculation. Also, this work adds a GPU version of the stencil computation, and the code could be run on a CPU or Cell BE cluster using MPI for parallelization.

II. Stencil and Application

The stencil considered in this paper is a 7-point stencil of the form

$$\begin{aligned} u_{ijk}^{(n+1)} \quad = \quad & {}^{(1)}c_{ijk}\left(u_{ijk}^{(n)}\right)^2 + {}^{(2)}c_{ijk}u_{ijk}^{(n)} + {}^{(3)}c_{ijk} + \\ & {}^{(4)}c_{ijk}u_{i+1,j,k}^{(n)} + {}^{(5)}c_{ijk}u_{i-1,j,k}^{(n)} + \\ & {}^{(6)}c_{ijk}u_{i,j+1,k}^{(n)} + {}^{(7)}c_{ijk}u_{i,j-1,k}^{(n)} + \\ & {}^{(8)}c_{ijk}u_{i,j,k+1}^{(n)} + {}^{(9)}c_{ijk}u_{i,j,k-1}^{(n)} \\ & (n = 0, 1, 2, \dots) \end{aligned} \tag{1}$$

with coefficients ${}^{(\ell)}c_{ijk}, \ell = 1, 2, \dots, 9$, which are constant in time, but not constant in space. The upper indices (n), $(n+1)$ indicate the time step, while i, j, k are the spatial grid coordinates.

This type of stencil arises in the Finite Volume discretization of Penne's "Bioheat" equation [11], the parabolic PDE

$$\rho C_\mathrm{p} \frac{\partial T}{\partial t} = \nabla \cdot (k \nabla T) - \rho_\mathrm{b}\omega_\mathrm{b}C_\mathrm{b}(T - T_\mathrm{b}) + \frac{\sigma}{2}\|\mathbf{E}\|^2 \tag{2}$$

used to model the temperature distribution T within (a part of) the human body. The simulation based on this equation is an essential part of hyperthermia cancer treatment planning [10]. In this treatment, the tumor is heated up to approximately $41°\mathrm{C}$, which makes it more susceptible to both radio and chemo therapies. In fact, heat is the most powerful sensitizer known to date [16]. Therefore, it is usually used as a complementary therapy. The heating process is done using non-ionizing radiation (microwaves). The aim is to create a constructive interference at the tumor location while avoiding hot-spots in the healthy tissue to minimize tissue damage. In equation (2), $\mathbf{E}$ models the electric field, ρ, C_p, k, ω, σ stand for physical material properties, the index b denotes a blood property.

* This work was supported by the Swiss National Science Foundation under grant $200021-117745/1$ and by an IBM Faculty Award on "Modeling, Simulation and Optimization in Hyperthermia Cancer Treatment Planning"

[1] High Performance and Web Computing Group, Dept. Computer Science, University of Basel, Switzerland
[2] Tech-X Corporation, Boulder CO, USA
[3] IT'IS Foundation, ETH Zurich, Switzerland

III. Hardware Architectures

The stencil code has been implemented on different architectures, which are briefly described in this section before detailing some of the algorithms that have been applied to improve performance.

A. The STI Cell BE Processor

The Cell BE was designed in a joint effort of Sony, Toshiba, and IBM. The novel processor is the core of Sony's Playstation 3, but it is also used as a high performance computing solution in IBM's Cell BE Blades.

The Cell BE's approach to achieving the high performance with a theoretical peak at 230 single precision GFlop/s is, contrary to the approach of recent CPUs, to provide eight specialized compute workhorses ("Synergistic Processing Elements", SPEs) that are controlled by a PowerPC processor, the PPE. The SPEs are composed of a "Synergistic Processing Unit" (SPU), which is a simple dual-issue, statically scheduled SIMD core, a local memory ("local store"), and the memory flow controller (MFC).

Both the PPE and the SPEs are simple processors in the sense that they are in-order RISC processors with a fixed-width instruction format. Forgoing branch prediction and out-of-order logic means freeing silicon space in favor of transistors dedicated to computation.

Each SPE contains 128 SIMD registers of 128 bits width. This means that a register could contain four single precision floating point data elements or two double precision data elements. Operands are always 128 bit words. Operations are therefore carried out in SIMD fashion. Depending on the datatype the SPEs perform 2-way, 4-way or 8-way SIMD operations.

Running at 3.2 GHz and supporting a fused multiply-add operation, the SPE's theoretical single precision peak performance is 4 units $\cdot\, 2$ Ops/unit $\cdot\, 3.2$ GHz $= 25.6$ GFlop/s or 12.8 GFlop/s for double precision. (Note that the first generation of Cell BE chips only reached 1.8 GFlop/s double precision performance because the corresponding operations requiring 13 cycles would stall the 7-stage pipelined SPE for 6 cycles.)

While the PPE features a conventional cache hierarchy, there aren't any caches on the SPEs, and each SPUs can only access the data residing within its 256 KB of local memory, i.e. it is not possible to access main memory directly from an SPE. The memory flow controller provides coherent data transfers between the main memory and the local memories, or between the local stores of the individual SPEs. Memory accesses have to be done by explicit DMA commands. The DMA transfers work asynchronously. Thus the access latencies can (and should) be hidden by computation. In fact, each SPE has two instruction pipelines for calculation and I/O commands, respectively.

On chip, the data is transferred over the "element interconnect bus", which provides high communication bandwidth with a peak of 204.8 GB/s. The bus is organized in a ring structure; it consists of 4 data rings, 2 rings running clockwise, and 2 rings running counterclockwise. The element interconnect bus is the network connecting the SPEs and also the interface to main memory providing a bandwidth of 25.6 GB/s.

B. NVIDIA's GeForce 8800/Tesla Computing Solution

With the G80 series, NVIDIA has launched a product line of GPUs that could be easily used for general purpose computation (GPGPU). The Tesla S870 computing system used for our benchmarks consists of four GeForce 8800 GTX GPUs and is endowed with 6 GB of DDR3 RAM (1.5 GB on-board per GPU).

The GeForce 8800 GTX GPUs feature 8 texture processor clusters, each containing 2 so-called "streaming multiprocessors", which are in fact 8-way SIMD units. In other words, there are 128 scalar arithmetic units. Each multi-processor is endowed with a small amount of on-chip local memory, which, with a size of 16 KB, is even smaller than the local store of the Cell BE. However, there is a large register file containing 8192 registers for each multi-processor. The bandwidth to the on-board DRAM is as high as 86 GB/s, while the access latencies are in the range of hundreds of cycles. To hide the latencies, the GPU is designed to potentially run millions of threads. In contrast to CPU threads, GPU threads are very lightweight, and context switching is virtually free.

The GPUs run at a clock speed of 1.35 GHz. A fused multiply-add operation is supported, thus the theoretical single-precision peak performance of the Tesla system is 4 GPUs $\cdot\, 128$ ALUs/GPU $\cdot\, 2$ Ops/ALU $\cdot\, 1.35$ GHz $=$ 1382.4 GFlop/s. (Note that the system supports only single precision operations. However, with the new Tesla C1060 and Tesla S1070 models, NVIDIA has made 64-bit precision GPU systems available.)

In parallel to the hardware, a language, CUDA [9], or rather a programming model and API, has been developed by NVIDIA that abstracts the graphics hardware. Its purpose it to serve as a general purpose programming language for GPUs. As a language, CUDA is a slight extension of ANSI C. The GPU is invoked by executing a "kernel" function that is run by potentially thousands of GPU threads in parallel.

IV. Algorithmic Considerations for the Cell BE

An essential prerequisite for maximizing performance is to avoid stalls of the computation units that are caused by waiting for input data on which the computation is executed. Ideally, the data movement would occur concurrently with the computation, i.e. data transfer latencies are hidden with computation.

On an architecture with explicitly managed memory such as the Cell BE, double or multibuffering is a common technique to achieve this [1]. Conceptually, two or more data buffers are used; one set of buffers is receiving data while on another set the computation is performed.

Unfortunately, for stencil codes, if implemented in the obvious manner, the arithmetic intensity (i.e. the Flop per transferred byte ratio) is so low that double buffering alone does not

result in a well-performing code. Clearly, the performance of such a code is bandwidth-limited. In order to increase the Flop per byte ratio the structure of the algorithm has to be modified. We use blocking in the temporal dimension, i.e. the data is kept in local memory as long as possible for re-use. This is done in an explicit manner (as opposed to a cache-oblivious method [6]), since the amount of available memory is known and no further data structure is needed for bookkeeping.

To calculate an upper bound for the performance if no time blocking is used, we could assume that the entire grid of solution values is loaded simultaneously into the local memory. This means that per grid point we have to load one solution value, while the neighbor solution values are assumed to be readily available, and 9 coefficients for the stencil considered here. After the computation, one data element is written back to main memory. The stencil considered carries out 16 floating point operations per grid point. As the Cell BE could provide 25.6 GB/s of bandwidth to main memory, for single precision data we could expect at most

$$P_{\max} \leq 25.6 \text{ GB/s} \cdot \frac{16 \text{ Flop}}{(1+9+1) \cdot 4 \text{ B}} = 9.31 \text{ GFlop/s}.$$

A. Spatial Blocking

The local stores of the SPUs are too small to hold all the buffers required for real-world problems. Therefore, the data must be partitioned into smaller chunks that fit into local memory. The way of decomposing the data that allows for maximum data reuse is depicted in Fig. 1. The data cube is split into planes along the x axis. Each plane is subdivided into panels along the y axis. As in [5], we decided not to split the data in z direction, which is the unit stride direction. This simplifies data transfers because contiguous chunks of data could be loaded from main memory.

The algorithm proceeds panelwise in x direction before the neighboring panel in y direction is selected. This allows to reuse previously loaded input data: in order to compute the stencil on the inner points of a panel, three input panels are required, namely the current panel as well as the panels in front of and behind the current panel are required. Therefore, when proceeding to the next panel in x direction, two of the panels could be reused while the data in the first of the three panels is discarded. If the y direction were prioritized over the x direction only one line of data could be reused.

Since we allow non-constant coefficients in space, panels of coefficient data have also to be loaded. Note that, since the result value $u_{ijk}^{(n+1)}$ depends on its nearest neighbors, each panel of solution input data must contain two extra lines to the left and to the right. However, the coefficients are required only on the inner grid points.

Another advantage of this decomposition scheme is suitability for parallelization. Domain decomposition cuts along the x and y axis are straight forward to implement. Cutting along the z axis is neglected. We have found that cutting only along the x axis yields the best performance.

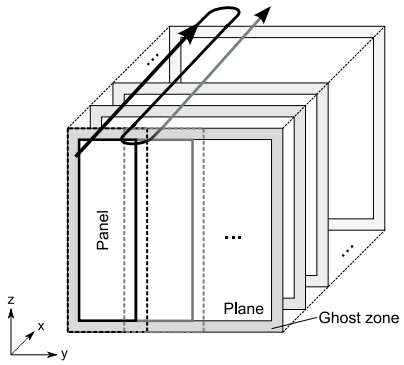


Fig. 1. Spatial decomposition of the data for the Cell BE. The arrow indicates the direction in which the data is processed.

B. SIMDization

In order to take full advantage of the Cell BE's computational power, the code must be vectorized. The SPUs are SIMD units that operate on 128 bit registers and therefore could carry out 4 single precision or 2 double precision floating point operations at once.

For the stencil code, this implies that four values in unit stride direction, i.e. along the z axis, are calculated at once. However, since SIMD operations are restricted to input data elements that are a multiple of 128 bit apart, the data has to be re-aligned so that the operation $u'_{i,j,k} \leftarrow \alpha u_{i,j,k-1} + \beta u_{i,j,k+1}$ could be performed. The `spu_shuffle` intrinsic is used to do that. It allows to arbitrarily "mix" two adjacent 128 bit vectors byte by byte.

Since the use of the shuffle operation introduces a potential computation bottleneck, it is only applied when absolutely necessary: in non-unit stride directions it could be avoided by adding paddings so that the number of grid nodes in y and z direction becomes divisible by 4.

C. Double Buffering

The decomposition method and the choice to process the panels in x direction in the inner loop, suggests a double buffering scheme. Besides the three buffers of panel data that are used for calculation, a fourth buffer is needed in which the data for the following computation is preloaded. The handling of the coefficients follow the regular double buffering scheme: there are two sets of coefficient buffers, one being used for the current calculation while the other set receives the data being preloaded.

This could be viewed as a pipeline, which is depicted in 2. The pipeline time proceeds along the horizontal axis. The arrows indicate data dependencies. In the first step, the coefficients used to calculate panel 1 are loaded ("ld c_1"), together with the input data on the panels 0, 1, and 2 ("ld $u_0^{(0)}$", "ld $u_1^{(0)}$", "ld $u_2^{(0)}$"). (The super-index (0) indicates that the data concerned is the zero-th timestep of the current iteration.)

In the next step the second coefficient set c_2 and the third panel of input data is preloaded into additional buffers because

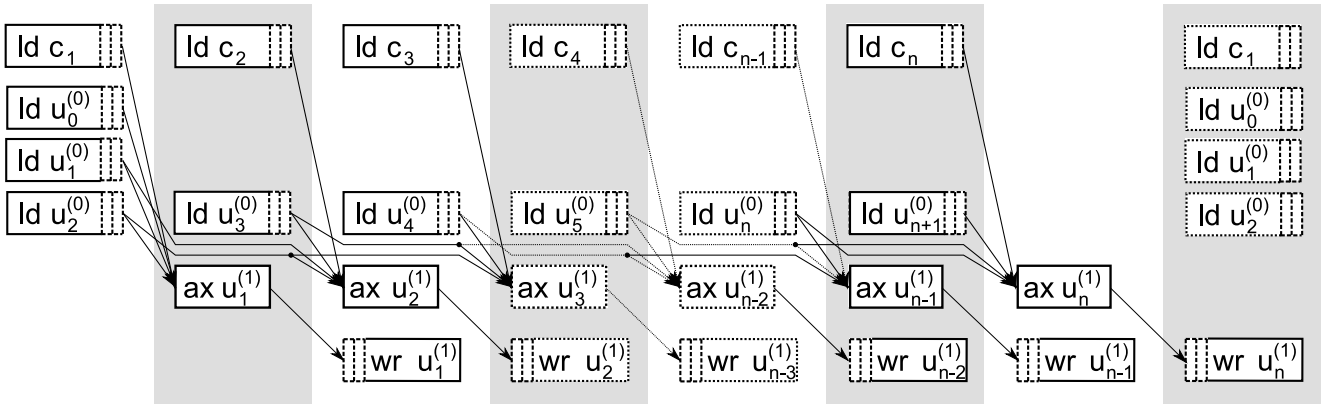


Fig. 2. The double buffering pipeline. "ld $u_i^{(0)}$" means that the 0th timestep (i.e. the initial values) of panel i is loaded, "ax $u_n^{(1)}$" stands for the computation of the data in panel i ("assess $u_n^{(1)}$"), while "wr $u_i^{(1)}$" denotes writing back of the result to main memory. The arrows indicate data dependencies.

c_1 and the buffers used for $u_0^{(0)}$, $u_1^{(0)}$, $u_2^{(0)}$ are still in use as indicated by the arrows ending at "ax $u_1^{(1)}$". This box symbolizes that the first timestep of panel 1 is calculated, which is done simultaneously while loading, i.e. computation and I/O is overlapped.

In the third pipeline step the pipeline has reached its full state. Data is loaded and computed as before. Additionally, the data computed in the previous step could be written back to main memory ("wr $u_1^{(1)}$") simultanously with the computation of $u_2^{(1)}$.

When the last panels in x direction are reached, the pipeline again requires two steps to drain. Note that in the last step used to write back the solution data of the last computed panel the input and coefficient data for the next panel block in y direction could already be preloaded.

D. Temporal Blocking

The pipeline described above could be extended so that in each pipeline step multiple stencil timesteps are performed. This allows to reuse computed data for a further stencil iteration in the same pipeline step without having to write the intermediate data back to main memory. This, of course, is only feasible if none or not all of the intermediate results are of interest. This method, referred to as "circular queue" [5] dramatically reduces memory traffic. However, the local memory requirements increase because data (computed data and coefficients) have to be kept in local memory for several pipeline steps.

Fig. 3 shows the filling phase of the pipeline. For clarity, the loading of the coefficients has been omitted in the diagrams. The diagrams demonstrate 3-stage time blocking as an example, i.e. per iteration 3 stencil applications are performed.

As in the double buffering case, 3 panels (along with the first set of coefficients) need to be loaded until the first calculation step could be performed, which could be overlapped with the data transfer loading the fourth panel (and the second set of coefficients). In the third step the calculation result $u_2^{(1)}$ is used to calculate the second timestep on the first panel $u_1^{(2)}$.

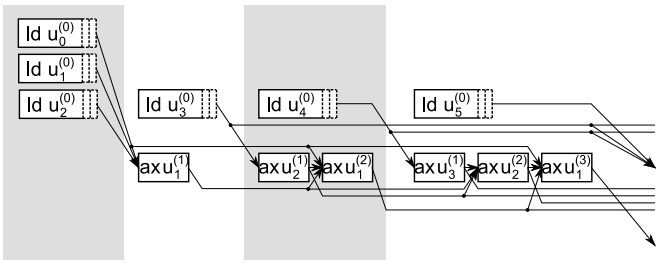


Fig. 3. The filling phase of the time blocking pipeline

The diagrams are "wirings" for PDEs with Dirichlet boundary conditions. Hence, the required inputs for $u_1^{(2)}$ are $u_0^{(1)} = u_0^{(0)}$, $u_1^{(1)}$, $u_2^{(1)}$.

Note that the first write back occurs only in the 5th pipeline step in Fig. 4, or in the $(B+2)^{\text{nd}}$ step, respectively, if B is the "time blocking factor", the number of stencil applications per iteration.

Also note that double buffering is in fact a special case of the circular queue method for temporal blocking with $B = 1$.

For B-stage temporal blocking, $3(B+1)$ buffers are required to hold the solution data as could be seen in a step of the pipeline's working phase:

- one buffer has to hold the panel preload data $u_{i+2}^{(0)}$,
- 3 buffers are required as input data to compute the first timestep in the sequence, i.e. the first stencil application on the input data, $u_{i-1}^{(0)}$, $u_i^{(0)}$, $u_{i+1}^{(0)}$,
- Each of the $B-1$ following timesteps require one buffer to which the result is written, $u_{i-t+1}^{(t)}, t = 1, 2, \ldots, B-1$, as well as two buffers $u_{i-t-1}^{(t)}$, $u_{i-t}^{(t)}$ needed to compute the next timestep in the next pipeline step,
- In the last timestep only two buffers are required, one that receives the data from the current calculation $u_{i-B+1}^{(B)}$ and one for $u_{i-B}^{(B)}$ serving as source for the write-back to main memory.

For the non-constant coefficient stencil in addition to the solution buffers the coefficient buffers are required. Since the

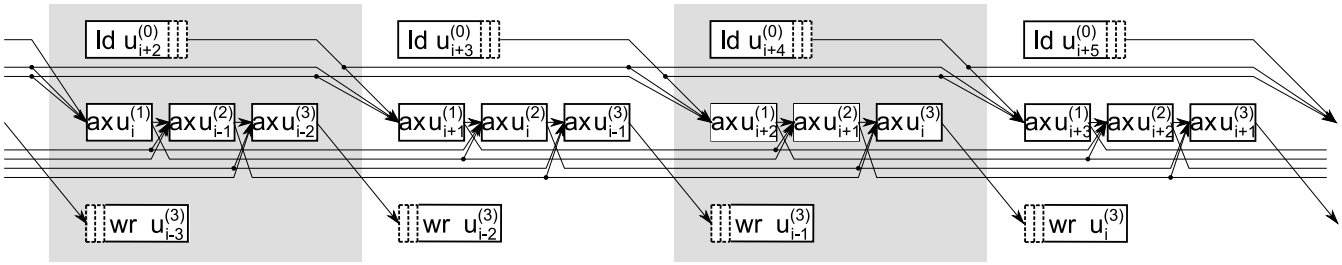


Fig. 4. The working phase of the time blocking pipeline

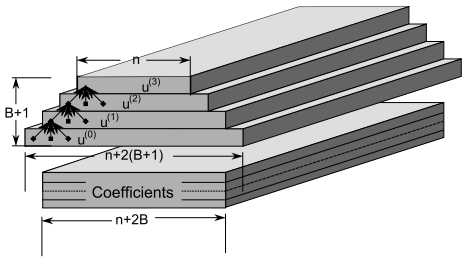


Fig. 5. Panels holding solution data and number of lines per panel. The trapezoid shape is caused by the data dependencies symbolized be the small arrows to the left.

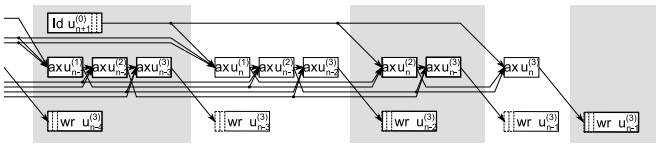


Fig. 6. The draining phase of the time blocking pipeline

computation of timestep t lags behind in space by t panels, B sets of coefficients are required, and an additional one used to preload the coefficients for the next pipeline step.

Note however, that, because of the data dependencies, the panel sizes vary. The number of computable lines is reduced by 2 per blocked timestep. Let m be the number of inner lines in the last timestep B of a phase. Then the panels assigned to timestep $B-1$ must hold $m+2$ lines, etc. $m+2B$ lines are required for the coefficient panels. This is shown in Fig. 5. Hence,

$$\begin{aligned} & 4(m+2B) + 3\sum_{i=1}^{B-1}(m+2i) + 2m + \\ & +9(B+1)\,(m+2(B-1)) \\ = \quad & 12(B+1)m + 21B^2 + 5B - 18 \end{aligned}$$

lines are required to be stored simultaneously in the local store.

Let n_x, n_y, n_z be the number of inner nodes in the respective direction. If the maximum number of lines fitting into the local store is C (which, of course, is dependent on the line length, $C = \lfloor Z/n_z \rfloor$ if Z is the available local store space), m is limited from above by

$$m \leq \frac{C - 21B^2 - 5B + 18}{12(B+1)}.$$

Note that the coefficients constitute a large part of the data. If the coefficients are constant in space, as many as $m \leq \frac{C-3B^2-5B}{3(B+1)}$ inner lines per panel could be used, i.e. $^7/_4$ times more than in the non-constant case.

The major drawback of the circular queue method is the fact that redundant computation is performed, which increases with increasing B (the "steps" at the panel boundaries in Fig. 5 are computed several times). To minimize redundant computation the number of lines m should be therefore as large as possible. Specifically, $\frac{1}{2}(N_y - 1)B(B+1)n_x n_z$ redundant stencil computations are performed per pipeline step, where $N_y := \lceil n_y/m \rceil$ is the number of panels along the y axis. As opposed to this, $n_x n_y n_z B$ stencil computations are actually required.

The arithmetic intensity is roughly $\frac{16B}{44}$ Flop/B since per pipeline step B stencil applications are performed, each contributing 16 Flops per grid node. At the same time $(1+9)\cdot 4$ Bytes are loaded for solution data and coefficient and 4 Bytes are written per grid node. So the performance is bounded by

$$P_{\text{max}} \leq 25.6 \text{ GB/s} \cdot \frac{16B \text{ Flop}}{44 \text{ B}}.$$

For $B = 3$, for which we achieved best performance, e.g., $P_{\text{max}} \leq 27.9$ GFlop/s. Note that this calculation doesn't count the redundant stencil computations. The real performance will decrease with increasing number of redundant calculations.

E. Loop Unrolling

Formulating equation (1) as a sequence of fused multiply-adds (*fma*s) results in a long and skinny arithmetic expression tree, which means that the result of the previous computation is required to proceed with the next computation. On the Cell BE, fma has a 6 cycle latency, thus many stalls are incurred.

For the temporally blocked codes less bandwidth is required, so it is essential to remove these stalls in order to get good performance. This could be done with loop unrolling: several of the arithmetic expression trees are interleaved so that register loads and stores and arithmetic evaluations could occur in parallel (the SPUs feature two instruction pipelines to allow this; the "even" pipeline handles arithmetic operations, and the "odd" pipeline I/O-specific instructions).

V. Parallelizing With MPI

In order to solve large-scale problems fast, we have parallelized the code with MPI. Essentially the same MPI code

could be used for both traditional CPU clusters and Cell Blade clusters. Only the stencil kernel code has to be substituted for the respective architecture.

Algorithm 1

```
1:  for number of iterations do
2:    {send boundaries}
3:    for all neighbors do
4:      MPI_Irecv (recv_buf)
5:      copy boundaries to send_buf
6:      MPI_Isend (send_buf)
7:    end for
8:    compute stencil
9:    {process boundaries}
10:   while (id = MPI_Waitany) ≠ UNDEFINED do
11:     for i on boundary do
12:       result[i] ← result[i] + coeff[i, id] · recv_buf[i']
13:     end for
14:   end while
15: end for
```

The MPI code handles the domain decomposition and boundary data communication. The latter is implemented in a way that allows overlapping communication with computation [15] as shown in Algorithm 1. This procedure is executed in parallel by each of the MPI processes. The domain could be decomposed along all of the three axes. Each MPI process is assigned one of the subdomains.

In the first phase (lines 2 to 7) the boundary values are prepared for sending. The values are copied to a special send buffer because they will be modified in the second, the compute phase. A non-blocking receive (`MPI_Irecv`) is pre-posted that will receive the data transmitted by the non-blocking `MPI_Isend` command (of another process). After the compute phase, the processes are synchronized by waiting for the boundary data of any of the neighboring processes (line 10).

In order for the stencil code from the sequential CPU or the Cell version to be used without modifications, each subdomain must be endued with a layer of ghost nodes at the boundary that is initialized with zero values. If a stencil is evaluated at the subdomain boundary, the contribution in the direction of the boundary is cut off. It is added in the boundary processing phase in line 12. Note, however, that the temporally blocked version requires extra handling of the boundaries.

VI. GPU Implementation

So far, the GPU version of the stencil code has been implemented in a straight-forward manner without any algorithmic optimizations. Nevertheless, quite satisfactory performance results are obtained.

The domain is decomposed in x direction to share the workload among the GPUs. All the data within a subdomain are then copied to the respective GPU and kept there until all required stencil iterations have been performed. Between two iterations the subdomain boundary is exchanged similarly as in the MPI case by copying it back to the CPU system and distributing it to the neighboring GPUs (as there is no possibility for inter-GPU data communication).

The stencil sweep itself is then computed by one thread per grid point. To access the values on the neighboring grid points, textures are used instead of accessing global memory directly, because texture memory is cached. This is a major performance benefit.

VII. Experimental Performance Results

The performance benchmarks have been carried out on an IBM Blade Center QS22 operating two Cell BE chips (16 SPUs in total) and on an NVIDIA Tesla S870 GPU system. Note that we only present single precision results here.

Additionally, for comparison, a "traditional" system has also been used. For the MPI benchmarks, we used a cluster powered by Intel Xeon X5355 Clovertown quad-core CPUs running at 2.66 GHz and equipped with 2×4 MB of L2 cache. The fully buffered DIMM technology allows up to 21 GB/s of bandwidth. Some OpenMP-parallelized benchmarks of the sequential reference implementation have been carried out on the Intel Clovertown dual quad-core SMP machine for comparison with the Cell BE. In Fig. 7 the theoretical peak performances for single precision operations of the machines used are listed. The bandwidth lists the bandwidth available to the main memory, in case of the GPU to the CPU system, which is limited by the data rate of the PCIexpress bus. The bandwidth to the 1.5 GB of on-card memory however is a lot larger (86 GB/s).

All the codes have been compiled using the GNU C compiler, except for the GPU codes, which require a special compiler.

A. Cell BE vs. CPU Performance

As seen in Fig. 8, the stencil code running on a Cell BE blade displays a linear speedup up to 4 SPUs. When using more SPUs the bandwidth requirements exceed the available bandwidth causing the SPUs to stall. The left figure shows the performances of the non-time blocked version. Using 3-stage time blocking, the performance could be roughly doubled, while the scaling behavior is preserved as shown in the right bar chart. Since in the current implementation, the domain is not decomposed in z direction, the algorithm performs better for for small z dimensions, because more panel lines could fit into the local store and hence decrease redundant computations. For problem sizes with small z dimension, an absolute performance of 4.1 GFlop/s on 1 SPU and 20 GFlop/s on 16 SPUs was observed.

The performance of the CPU version is shown in the top left portion of Fig. 8. Note that this code, other than using the `-O3` compiler flag, hasn't been optimized. Datta et al. have shown in [5] that for Intel processors (Itanium 2), for the stencil with constant coefficients a speedup of $\approx 1.7\times$ over the naïve implementation could be obtained when using time skewing, which is another method for temporal blocking.

Architecture	*Sockets / Cores*	*Performance/core*	*Performance/socket*	*Overall performance*	*Bandwidth*
Intel Clovertown X5355	$1 \cdot 4$	18.66 GFlop/s	**74.64 GFlop/s**	74.64 GFlop/s	21 GB/s
IBM QS22	$2 \cdot 8$	25.6 GFlop/s	**204.8 GFlop/s**	409.6 GFlop/s	52.2 GB/s
NVIDIA S870	$4 \cdot 128$	2.7 GFlop/s	**345.6 GFlop/s**	1382.4 GFlop/s	16 GB/s

Fig. 7. Theoretical peak performances of the architectures used to perform the benchmarks

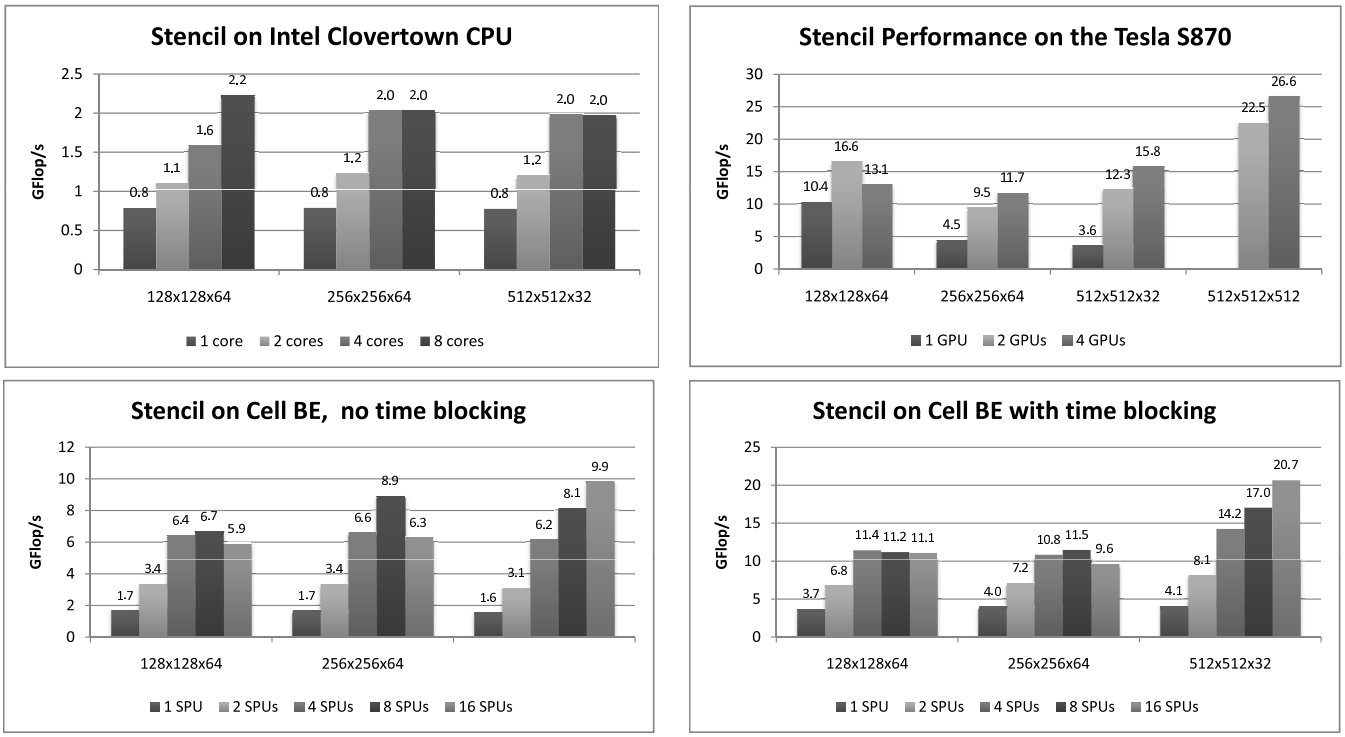


Fig. 8. Results of the performance benchmarks on the architectures used

The number suggest that we have a per core speedup (1 CPU core — 1 SPU) of $3\times$ and a per processor speedup (1 Cell BE processor — 1 Intel Clovertown X5355) of $6\times$.

B. GPU Performance

The performance achieved on the Tesla S870 GPU computing solution is shown in the top row to the right in Fig. 8. The large problem (512^3 grid points) displays a good performance of 22 GFlop/s on 2 GPUs (there was not enough memory to run the problem on a single GPU). This result has been obtained without any algorithmic optimization. GPU-specific code optimizations have been applied, though. The figure shows almost linear speedup when going from 1 GPU to 2 GPUs. The transition from 2 to 4 GPUs results in a slowdown due to limited bandwidth.

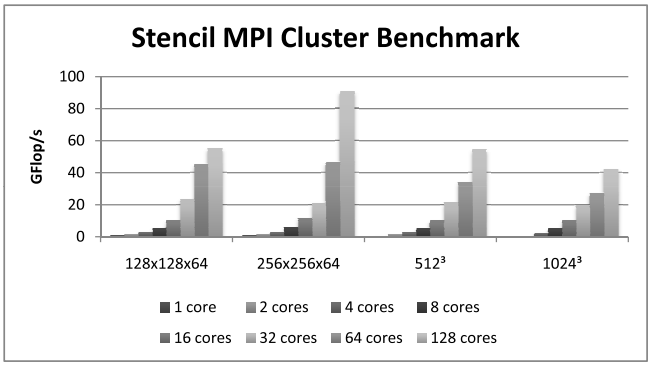


Fig. 9. Performance benchmark results on the CPU cluster. Parallelization was done using MPI.

C. MPI Results

The MPI code has been run on a CPU cluster. The code scales very well up to 128 processes on the CPU architecture. The figure demonstrates perfect scaling with up to 32 processes, which is due to the fact that the machine has 32 nodes. For up to 32 processes, only one core per node was used. For 64 and 128 processes, 2 and 4 cores per node were used, resulting in increased network traffic, incurring a relative performance decrease.

VIII. Related Work

Stencil performance on the Cell BE processor has been benchmarked in [17] and [5]. Using temporal blocking, performances of up to 65 GFlop/s for single precision stencil

computations are reported. The stencils considered are, however, somewhat simpler: the coefficients are kept constant both in time and space, which allocates more of the bandwidth and local store memory for the solution values.

Temporal blocking and tiling algorithms for stencil computations have been investigated and described in [6], [8], [12], [13], [14], [7], [18].

In [4], [3] GPUs are explored as accelerators for a computational fluid dynamics simulation. The method used to compute the simulation translate to stencil computations. Speedups of $29\times$ (in 2D) and $16\times$ (in 3D), respectively, over the Fortran reference implementation running on an Intel Core 2 Duo 2.33 GHz CPU are reported.

IX. Conclusion and Future Work

Stencil-based methods constitute an important class of numerical methods [2]. Unfortunately, the low arithmetic intensity for most stencil applications result in poor performance that is far from the machine's peak performance. On the Cell BE, the available bandwidth is the limiting factor if the stencil is implemented in the obvious way. Therefore, algorithmic optimizations must be applied to exploit the Cell BE's computational power. The same holds true for GPUs, although the high on-board bandwidth allows decent performance.

We have detailed a method of temporal blocking, to improve performance by considering the data locality. Other methods of temporal blocking have been proposed; the one chosen is easily parallelizable and naturally extends the double buffering scheme. So far, on a dual quad-core Intel Clovertown X5355 (8 cores) a performance of 2 GFlop/s was reached. On a Cell blade (16 SPUs) 22 GFlop/s could be measured while a single GeForce 8800GTX GPU provided 7.8 GFlop/s.

Adapting the MPI code to run on a Cell BE cluster is work in progress. Other blocking methods for both the Cell BE and GPUs will be investigated in the hope that a performance-wise more efficient method will be found. It is also planned to extend to prototypes to a framework supporting more general stencils.

Acknowledgments

The authors like to thank Hema Reddy from IBM, Austin, for her support and the IBM Systems and Technology Group, Poughkeepsie, New York for providing access to their Cell BE cluster and for their support. The authors also acknowledge Georgia Institute of Technology, its Sony-Toshiba-IBM Center of Competence, and the National Science Foundation, for the use of Cell Broadband Engine resources that have contributed to this research. Finally, we would also like to thank Philip Lüscher, Master student at the Computer Science Department, for porting the code to CUDA.

References

[1] Abraham Arevalo, Ricardo M. Matinata, Maharaja Pandian, Eitan Peri, Kurtis Ruby, Francois Thomas, and Chris Almond. Programming the Cell Broadband Engine™ Architecture: Examples and Best Practices. http://www.redbooks.ibm.com/redbooks/pdfs/sg247575.pdf, checked in 08/2008.

[2] Krste Asanovic, Ras Bodik, Bryan C. Catanzaro, Joseph J. Gebis, Parry Husbands, Kurt Keutzer, David A. Patterson, William L. Plishker, John Shalf, Samuel W. Williams, and Katherine A. Yelick. The landscape of parallel computing research: a view from Berkeley. Technical Report UCB/EECS-2006-183, Electrical Engineering and Computer Sciences, University of California at Berkeley, December 2006.

[3] Tobias Brandvik and Graham Pullan. Acceleration of a 3D Euler solver using commodity graphics hardware. In *In Proc. of 46th AIAA Aerospace Sciences Meeting and Exhibit, 7-10 Jan 2008, Reno, Nevada, USA*. In Press.

[4] Tobias Brandvik and Graham Pullan. Acceleration of a two-dimensional Euler flow solver using commodity graphics hardware. *Proceedings of the Institution of Mechanical Engineers, Part C: Journal of Mechanical Engineering Science*, 221(12):1745 – 1748, 2007.

[5] Kaushik Datta, Shoabib Kamil, Samuel Williams, Leonid Oliker, John Shalf, and Katherine Yelick. Optimization and Performance Modeling of Stencil Computations on Modern Microprocessors. *SIAM Review*, 2008. To appear.

[6] Matteo Frigo and Volker Strumpen. Cache oblivious stencil computations. In *ICS '05: Proceedings of the 19th annual international conference on Supercomputing*, pages 361–366, New York, NY, USA, 2005. ACM.

[7] Markus Kowarschik, Christian Weiβ, Wolfgang Karl, and Ulrich Rüde. Cache-aware multigrid methods for solving Poisson's equation in two dimensions. *Computing*, 64(4):381–399, 2000.

[8] Zhiyuan Li and Yonghong Song. Automatic tiling of iterative stencil loops. *ACM Trans. Program. Lang. Syst.*, 26(6):975–1028, 2004.

[9] NVIDIA. NVIDIA CUDA Compute Unified Device Architecture – Programming Guide. http://developer.download.nvidia.com/compute/cuda/2.0-Beta2/docs/Programming_Guide_2.0beta2.pdf, Checked in 08/2008.

[10] M. M. Paulides, J. F. Bakker, E. Neufeld, J. van der Zee, P. P. Jansen, P. C. Levendag, and G. C. van Rhoon. The HYPERcollar: A novel applicator for hyperthermia in the head and neck . *International Journal of Hyperthermia*, 23:567 – 576, 2007.

[11] Harry H. Pennes. Analysis of Tissue and Arterial Blood Temperatures in the Resting Human Forearm. *J Appl Physiol*, 1(2):93–122, 1948.

[12] Gabriel Rivera and Chau wen Tseng. Tiling optimizations for 3D scientific computations. In *In Proceedings of SC'00*, 2000.

[13] Sriram Sellappa and Siddhartha Chatterjee. Cache-Efficient Multigrid Algorithms. *Lecture Notes in Computer Science*, 2073:107–116, 2001.

[14] Yonghong Song and Zhiyuan Li. A compiler framework for tiling imperfectly-nested loops. In *In Proc. of 12th International Workshop on Languages and Compilers for Parallel Computing, (LCPC99*, pages 185–200. Springer-Verlag, 1999.

[15] Volker Strumpen and Thomas L. Casavant. Exploiting communication latency hiding for parallel network computing: Model and analysis. In *Proc. PDS'94*, pages 622–627. IEEE, 1994.

[16] J. van der Zee. Heating the patient: a promising approach? *Ann Oncol*, 13(8):1173–1184, 2002.

[17] Samuel Williams, John Shalf, Leonid Oliker, Shoaib Kamil, Parry Husbands, and Katherine Yelick. Scientific computing Kernels on the cell processor. *Int. J. Parallel Program.*, 35(3):263–298, 2007.

[18] David Wonnacott. Time skewing for parallel computers. In *Proceedings of the Twelfth Workshop on Languages and Compilers for Parallel Computing*, pages 477–480. Springer-Verlag, 1999.

Fast Cache Miss Estimation of Loop Nests using Independent Cluster Sampling

Kamal Sharma, Sanjeev Aggarwal, Mainak Chaudhuri, Sumit Ganguly
Computer Science and Engineering Department,
Indian Institute of Technology Kanpur, India
{kamals,ska,mainakc,sganguly}@iitk.ac.in

***Abstract*—We introduce independent cluster sampling, a compile-time sampling technique applied to the iteration space of perfectly nested loops for estimating the number of cache misses experienced by the loop body. In this study, we squarely focus on dense array accesses in nested loops and present statistical error analysis of our approach and detailed experiments on a number of popular loop kernels drawn from dense linear algebra and the SPEC95 suite executing on a range of cache organizations differing in capacity, associativity, and block size. The experimental results show that in most cases we achieve much better cache miss estimates compared to other popular sampling techniques that work on full cache access traces. Since our technique does not require full cache access tracing or profiling, it speeds up the estimation process by more than a factor of hundred on complex loops compared to full cache simulations. Further, we show how our cluster sampling technique, when integrated in a compiler pass, can be directly applied to drive loop optimizations such as loop permutation and tile size determination.**

***Index Terms*—Performance measurement, memory hierarchy, cache miss estimation, iteration space sampling, loop optimization.**

I. Introduction

DRAM latency continues to be the most prominent bottleneck in server, desktop, and mobile computing systems. A large number of memory hierarchy optimization techniques has been proposed for loop nests. The analytical techniques like integer liner programming methods, linear Diophantine equations and first order logic formulation for estimating the number of cache misses in a loop nest. Full-scale profiling and trace-driven cache simulation techniques are also used for optimizing large complex programs. However, this technique often turns out to be ineffective for highly associative caches either due to significant complexity of the logic formulas or due to time consuming nature to have a full profile run.

In this paper, we introduce a source-level independent cluster sampling technique for fast estimation of the number of cache misses. As a first step to demonstrate its effectiveness, we focus on perfectly nested loops with dense array accesses. The technique draws clusters of consecutive samples from the dynamic iteration space of the loop. Each sample is an array access coming from the loop body. By sampling a cluster of consecutive dynamic accesses our technique accounts for spatial locality (if any) in the loop body. Multiple such sampled clusters try to capture the temporal locality in the access stream. While speeding up the estimation process by more than two orders of magnitude compared to full cache trace profiling on complex loops, our technique offers excellent cache miss estimates for extremely large datasets across a large number cache organizations differing in capacity, block size, and associativity. We compare our technique against four previously proposed sampling techniques that draw samples from a full trace of cache accesses. The most attractive aspect of our technique is that it does not require the full access trace and can be easily integrated into a compiler pass. To demonstrate the usefulness of this technique, we show that it can be successfully applied to identify the optimal permutation of a loop nest (leading to subsequent generation of the necessary unimodular transformation) and to determine the tile size of a tile-transformed loop as a part of the compilation process itself.

In the next section, we discuss the related work. Section II presents some background terminology and formalizes some of the assumptions. Section III outlines our independent cluster sampling approach. A mathematical analysis for our approach is given in Section IV, which identifies the key parameters and offers an estimation error bound of our approach. Section V discusses detailed experimental results on several important loop kernels drawn from linear algebra and the SPEC95 suite. We conclude in Section VI.

A. Related Work

There have been numerous techniques proposed for cache miss estimation. We broadly classify these into three categories, namely, analytical methods, sampling, and full simulation.

Analytical methods: Prior research in this area deals with the precision in calculating cache miss behavior. The concept of cache miss equations (CME) was introduced by Ghosh et al. [9]. This technique used linear Diophantine equations to calculate the number of cache misses. First order logic formulas in the form of Presburger arithmetic were used by Chatterjee et al. [4] to count the exact number of cache misses in loop nests. Both these methods use affine references for modeling cache misses. As the loop bounds and cache associativity increase, these approaches become unusable due to the complexity of the equations. These constraints are avoided in our independent cluster sampling algorithm. However, the cluster sampling scheme does not give the exact estimates of the number of cache misses.

The proposal by Fraguela et al. [7] creates probabilistic equations based on the data reuse between the references. This scheme uses probability factor for affine references to

approximate the misses. An added benefit that this work offers is to estimate the number of cache misses without knowing the base addresses of the data structures. However, the problems of affine references and error range bound persist.

The systems proposed by Vera et al. [18] and Vera and Xue [17] use simple sampling techniques in conjunction with cache miss equations (CME) to estimate the number of cache misses of whole programs. The complexity of generating numerical expressions for the data references still remains. This is in contrast to the ease of using our scheme. However, in this work we focus on loop nests only, and leave the extension to handle whole programs to future work.

Sampling: Most of the studies in this domain use memory access traces for estimating the cache miss ratios. All of the proposed schemes require full execution of the program. Collecting traces of large programs becomes a time-consuming task. These techniques are normally used for architectural studies.

The concept of set sampling on memory traces was introduced by Laha et al. [12]. The work used the sets where a past access was present in a cluster of references.

A theoretical bound on unknown references (first reference to a cache set) that are present in a cluster of references was proposed by Wood et al. [20]. The unknown references were estimated as misses using the life time of the known reference cache sets. To avoid errors, Kessler [11] proposed to always use trace samples large enough to fill half the cache sets. Our model ignores the unknown references and considers only full sets similar to the method introduced by Laha et al. [12]. However, our scheme does not require entire program execution to collect sampled traces.

Another proposal by Fu and Patel [8] to avoid the unknown references is to simulate certain percentage of instructions as cache warmup and use the rest of the references to predict the cache behavior.

Techniques proposed by John W. Haskins and Skadron [10] try to reduce the amount of warmup references. Choosing the appropriate sample length is important for accuracy and it depends on the kind of stream generated by the application. Eeckhout et al. [6] try to solve this problem by determining the optimal sample length. However, it requires one complete pass over the benchmark.

To avoid storing the entire trace, checkpointing techniques over the trace are introduced by Wunderlich et al. [21]. The proposal by Wenisch et al. [19] further optimizes the storage space of the checkpoint library by using a reduced set of states in the simulation window. Nevertheless, storing the states still requires a large amount of storage.

Other approaches use representative phases of the entire program to summarize the execution behavior. SimPoint, introduced by Perelman et al. [14], is one such tool, which partitions the program into phases based on basic block behavior by profiling.

In contrast with all these techniques, our scheme does not require any profiling of the application.

Full Simulation: Cascaval and Padua [3] developed a stack histogram scheme for estimating the number of cache misses. An entire pass is made on the data for a fully associative cache and then approximate estimates are made for set-associative caches. However, the scheme tends to produce large errors as the associativity increases beyond two.

These approaches are time-consuming. Our method offers significant advantage over these schemes in terms of the time required to carry out the estimation process.

II. Assumptions and Background Terminology

In this work, we consider perfectly nested loops with dense array accesses. The array sizes and the loop bounds are assumed to be known in advance. Further, the iteration space is assumed to be lexicographically ascending with unit step (standard transformations are available for converting any ascending iteration space to have unit step). Each array in a program is assumed to be allocated contiguously in virtual memory (this is true for statically allocated arrays and often true for dynamically allocated arrays). We consider only virtually indexed caches in this work (this holds true for the L1 caches in all high-end microprocessors today). An LRU replacement policy is assumed throughout for cache replacement.

A. Array Reference Mapping

For each array reference, we can define a mapping function that generates a <set value, wrap value> pair. The set value denotes the set number of the cache which the reference maps to (notice that the generation of this value assumes the knowledge of the cache indexing function). The wrap value is the number of times the array has wrapped around the cache leading to the current reference. In each program we assign some serial order of allocation to the arrays and assume that an array starts at set 0 of wrap w, where the previous array in the allocation order ends within wrap $w-1$. Notice that the <set value, wrap value> pair uniquely defines an array reference relative to a particular cache topology, given the starting wrap value of the array.

As discussed in Parker [13], the mapping function can choose row-major or column-major reference pattern based on the programming language. We choose the former as C programs are considered in this work.

B. Cache Miss Classification

In this work, we adopt a simple approach of classifying the cache misses into two categories, namely, cold misses and interior misses. Cold misses occur when a cache set is accessed for the first time (see Ghosh et al. [9]). Interior misses occur due to conflicts and capacity limitations in cache (see Chatterjee et al. [4]). An access to an already occupied cache set results in an interior miss if the currently held block and the accessed block are different.

III. Independent Cluster Sampling Algorithm(ICS)

Sampling is widely used to succinctly capture some property of a large population. However, cache miss estimation cannot

be directly transformed into a simple sampling problem because the probability of a reference causing a cache miss is inherently dependent on the past cache references which define the current cache contents. To take into account this temporal effect, we employ ICS for counting cache misses. In ICS, the samples are picked from a number of (temporally) localized regions in the population.

Each array reference in the program gets translated into a pair *(s,e)* where s is the set number and e is wrap value based on the mapping function. A miss is generated in a direct-mapped cache when two references have the same set value but different wrap values. In the case of a K-way set-associative cache, a cache miss occurs when the current wrap value is not present in the latest distinct K references to a particular cache set.

In our model, instead of looking at the whole stream of *(s,e)* pairs, subsequences of this stream are sampled for counting misses. These subsequences will be referred to as clusters or blocks (not to be confused with cache blocks).

A. Direct Mapped Cache Algorithm

The cache miss estimation algorithm takes as input a sequence STR of references of the form (s, e), where, $s \in S$ is a cache set value and e is a wrap value. The set S is the set of all cache set values. Each pair in STR is assigned a unique index in the temporal order of its appearance. The substream STR_{s_0} defined by a given set value $s_0 \in S$ is the subsequence of references of the form $(s_0, *)$. A *segment* of the substream STR_s is a maximal contiguous sequence of occurrences in STR_s, all of which have the same e-value. A segment is represented as $[a, b]$, where a is the starting index of the segment in STR and b is the final index of the segment in STR. The number of misses for STR_s is defined as the number of segments in STR_s and is denoted as M_s. Our goal is to estimate

$$M = \sum_{s \in S} M_s = \text{ sum of misses over all cache sets.}$$

Let C be a parameter which will be used to denote the cluster size. Assume that the length of the sequence m of references is known. Pick a random position u uniformly from from $1, \ldots, m$. Consider the random subsequence B of length $l = \min(C, m - u + 1)$ starting at u. Define the following random variable, for each cache set s in a direct-mapped cache.

$$X_s = \sum_{\substack{[a,b] \text{ is a non-first segment for } s \text{ in } B_s \\ [a,b] \text{ preceded by } [a',b'] \in B_s}} \frac{m}{C - (a - b')}$$

The final estimate is

$$X = \sum_{\text{cache-set } s} X_s \ .$$

Thus, X_s considers each segment $[a, b]$ that is not the first segment corresponding to cache set s in B_s. If $[a', b']$ is its preceding segment in B_s, a contribution of $\frac{m}{C-(a-b')}$ is added towards $[a, b]$.

For cache set s, denote by $M'_{s,C}$ the number of segments $[a, b]$ in the substream STR_s, such that its immediately preceding segment (for the same cache set) $[a', b']$ satisfies $a - b' < C$. Let

$$M'_C = \sum_s M'_{s,C} \ .$$

The above algorithm counts 0 for any segment whose preceding segment is at a distance of C or larger from it. The algorithm produces M'_C as the final estimate.

B. Set Associative Cache Algorithm

Consider the sequence of accesses of all elements that map to the same cache set. For a given memory address a, an *LRU*-segment segment for a is an interval $[i, j]$ of the sequence, such that the position i corresponds to a being loaded in the cache and j corresponds to an eviction of the item from the cache or j is the end of the sequence. Thus, $[i, j]$ represents one of the lifetimes of a in the cache.

Property 1. Suppose position i is occupied by address a. Then, i is the start of an *LRU*-segment iff the first k distinct occurrences of items prior to position i do not include a. Define $f(i)$ as follows. Suppose i is occupied by a. Going backwards from i, look at the first k distinct occurrences of addresses prior to position i. If a does not belong to these k addresses, then, i is the start position of an *LRU*-segment and $f(i)$ is the address of the kth distinct occurrence backwards from i. Otherwise, i is not the start position of an *LRU*-segment and $f(i)$ is undefined.

If the sequence of references $[f(i), i]$ belongs to the sampled cluster, then, i is correctly discovered as the start of an *LRU*-segment and otherwise it is not.

Let C be the length of a cluster block and m be the length of the entire address sequence (trace) across all cache sets. Suppose i is the starting point of an *LRU*-segment. The probability that i is discovered as the start of an *LRU*-segment is that the segment $[f(i), i]$ lies wholly in the cluster, which is,

$$\frac{C - (i - f(i) + 1)}{m - C + 1} \ .$$

The probability is 0 if $C < (i - f(i) + 1)$.

a) Algorithm.: Choose a random starting position s between 1 and $m - C + 1$ with equal probability and consider the cluster defined by $[s, s + C - 1]$. For each *LRU*-segment $[i, j]$ such that $f(i)$ lies within the cluster, form the estimate as follows.

$$Y = \sum_{\substack{\textit{LRU}\text{-segment}[i,j] \in \text{ cluster} \\ f(i) \in \text{cluster}}} \frac{m - C + 1}{C - (i - f(i) + 1)}$$

Finally, form t independent estimates $Y_1, \ldots, Y_t$ and return their average (or median of averages).This is the final estimate for the cache misses.

C. Complete Algorithm

The above algorithm presented count the corresponding misses for a given cluster of references. For a given stream, the miss distance between the sets may be too large. Thus, it is more rational to count the hits and misses of sets rather

than counting only misses. The same algorithm is used to count the hits. The final estimate of the misses is based on the corresponding proportion of hits/misses occurring in the cluster. Algorithm 1 presents the complete algorithm.

Algorithm 1 ICS Algorithm

1: $count\_hit \leftarrow 0$, $count\_miss \leftarrow 0$
2: $hit\_bound \leftarrow 0$, $miss\_bound \leftarrow 0$
3: **for** clusters from 1 *to* n_u **do**
4: **for all** reference <s,e> of cluster **do**
5: Update_cache(reference)
6: **if** reference is hit **then**
7: update $M'_{s,H}$
8: count_hit++
9: $hit\_bound \leftarrow hit\_bound+$ $check\_hit\_bound(reference)$
10: **else if** reference is miss **then**
11: update $M'_{s,C}$
12: count_miss++
13: $miss\_bound \leftarrow miss\_bound+$ $check\_miss\_bound(reference)$
14: **end if**
15: **end for**
16: calculate M'_C , M'_H
17: Reset_Cache()
18: **end for**
19: $M'_{C_{avg}} \leftarrow avg(M'_{C_1}, ..., M'_{C_{n_u}})$
20: $M'_{H_{avg}} \leftarrow avg(M'_{H_1}, ..., M'_{H_{n_u}})$
21: **if** count_hit $>$ count_miss **then**
22: $miss_estimate \leftarrow$m$-M'_{H_{avg}}$
23: **else**
24: $miss\_estimate \leftarrow M'_{C_{avg}}$
25: **end if**
26: **if** hit_bound $>$ 0.9*n_u*C **then**
27: $miss\_estimate \leftarrow M'_{C_{avg}}$
28: **end if**
29: **if** miss_bound $>$ 0.9*n_u*C **then**
30: $miss_estimate \leftarrow$m$-M'_{H_{avg}}$
31: **end if**
32: **return** miss_estimate

Steps 3 to 18 perform the basic counting of the hits/misses for the clusters in the stream. Based on the proportion of the hit/misses counted, appropriate miss estimation is provided (lines 21 - 25). The miss estimate for the counted hits is provided by subtracting the stream size m with the hits. Conditions in lines 26 - 31 are inserted to contain the upper bound errors as discussed in Lemma 2. The function check_(hit/miss)_bound returns a 1 if the corresponding distance between previous and current reference is 1, else it returns 0.

D. Applying the Algorithm

Figure 1 depicts a diagrammatic representation of the algorithm.

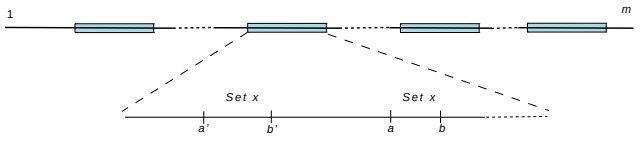


Fig. 1. Working of ICS

Each box represents a cluster of size *C*. The starting position of a cluster is randomly chosen. There are n_u such clusters sampled from the stream. We count the misses in each cluster using the technique discussed in the last section. The cache state is reset before the start of each cluster. Finally, an average is taken over the sampled clusters. Let us consider the following simple stream of *(s,e)*: *(2,1) (2,1) (2,1) (6,5) (7,8) (5,4) (2,3) (2,3)*. While counting misses for set 2 in a direct-mapped cache, *(2,1)* is the first segment. Thus, for reference *(2,3)* , $[a', b'] = [1, 3]$ and $[a, b] = [7, 8]$. The miss for set 2 is calculated as $\frac{m}{(C-4)}$.

For selecting the starting points of the clusters, one method is to pick random pairs from the stream and then sort them by their temporal order. However, this would consume time equivalent to sorting. Instead, we employ the method presented in [2] to generate sorted random numbers on the fly.

By looking at the above algorithm, one may get an impression that an entire pass over the program is necessary to collect the cache miss statistics. The uniqueness of our method is that it avoids full tracing or profiling. We implement the sampling technique in the compiler pass itself. Consider a perfectly nested loop of depth d with the k^{th} loop having lower and upper bounds of l_k and u_k, respectively. Let all the array references be part of the inner-most loop body and let there be t statements in the loop body. The k^{th} statement contains r_k distinct array references (notice that two references to the same array are considered distinct if they access different elements of the array). We define the order of references within a statement to be from right to left. The compiler first generates a stream of <set value, wrap value> of length $\prod_{p=0}^{d-1}(u_p - l_p) \sum_{q=0}^{t-1} r_q$, with each dynamic array reference representing one tuple. Next n_u clusters each of length C are sampled from this stream. Finally, for each cluster, the cache miss count is estimated using the already discussed technique. We show the cache miss estimation process with the help of a C-like pseudocode for matrix-matrix multiplication.

```
  // C - Cluster Size
  // ITER_PTS - number of clusters( n_u) in the
stream
  t1 = Random_sorted(ITER_PTS,ITER_SPACE);
  for( it = t1.next() ; it != -1 ; it = t1.next() )
  {
    for( l = it ; l <= (it + C) ; l++)
    {
      Map_iteration(l)->(i,j,k,reference)
      Update_cache(reference,i,j,k)
    }
   Reset_cache();
  }
```

`ITER_SPACE` refers to the entire stream of references and $t1$ is the set of sampled clusters with each cluster containing C consecutive <set value, wrap value> pairs. `Map_iteration(l)` maps each pair back to the unique array reference in the source loop, which is then used to determine if this reference results in a miss by looking up a simulated cache. Note that the simulated cache is invalidated at the end of each cluster.

IV. Mathematical Validation

In this section, we analyze the mean and variance of the random variable X defined in the last section and derive the

relative error in the estimate for a direct-mapped cache. The analysis for a set-associative cache is similar.

Lemma 1: $\mathsf{E}[X] = M'_C - |S|$.

Proof: Suppose $R = [a, b]$ is a segment corresponding to a cache set s such that R is not the first segment in the global stream for s.Recall that $|S|$ is the size of the cache set. Define an indicator variable y_w which is 1 provided either R or some prefix of w is not the first segment corresponding to s in the random block B and if the preceding segment of R, say $R' = [a', b']$, is at a distance of less than C. Define the set of segments

$$\textit{dense}_{s,C} = \{[a, b] \text{ is a segment for } s \mid \text{its preceding segment is } [a', b'] \text{ and } a - b' < C\}$$

Then,

$$X_s = \sum_{[a,b] \in \textit{dense}_{s,C}} \frac{m}{C - (a - b')} y_w$$

where, b' is the closing index of the previous segment of $[a, b]$ of cache set s. Thus,

$$\mathsf{E}[X_s] = \sum_{[a,b] \in \textit{dense}_{s,C}} 1 = \left|\textit{dense}_{s,C}\right| = M'_{s,C} - 1$$

where, the -1 appears since the algorithm always misses counting the first segment. Adding over all the s's gives the statement of the lemma. ∎

There are two parameters here, namely, the distance between successive segments for a cache set and the length C of block size. Both are set to C. However, the length of successive segments can be set to αC, for some constant $0 < \alpha < 1$, say $\alpha = 0.75$. The algorithm slightly changes as follows: if $b' - a > \alpha C$, we count 0 instead of the scaled expression. In this case, the expectation becomes

$$\mathsf{E}[X_s] = M'_{\alpha C} - |S| \ .$$

Next, we compute the variance of X. Define the statistic G as follows. In the following, for any segment $[a, b] \in \textit{dense}_{s,D}$, let b' denote the ending point of the immediately previous segment for the same cache set s.

$$G_{\alpha,C} = \sum \left\{ \frac{m(C - (\max(a_1, a_2) - \min(b'_1, b'_2)))}{(C - (a_1 - b'_1))(C - (a_2 - b'_2))} \right| [a_1, b_1] \in \textit{dense}(s_1, \alpha C) \text{ and } [a_2, b_2] \in \textit{dense}(s_2, \alpha C) \text{ and } \max(a_1, a_2) - \min(b'_1, b'_2) < C \Big\}$$

Note that the sum is taken over segments $[a_1, b_1]$ and $[a_2, b_2]$ of arbitrary cache set pairs s_1, s_2, such that (i) the end point b'_1 of the predecessor segment of $[a_1, b_1]$ is within a distance of αC from a (and analogously for the end-point b'_2 of the predecessor segment of $[a_2, b_2]$), and, (ii) C is large enough so that there exists a block of length C to include b'_1 and a_1, and, b'_2 and a_2. The latter is equivalent to saying that $C > \max(a_1, a_2) - \min(b'_1, b'_2)$.

Lemma 2: $\mathsf{Var}[X] \leq G_{\alpha,C}$.

Proof: We will calculate $\mathsf{E}[X^2]$.

$$\mathsf{E}[X^2] = \mathsf{E}\left[\left(\sum_s \sum_{[a,b] \in \textit{dense}_{s,\alpha C}} \frac{m}{C - (a - b')} y_{[a,b]}\right)^2\right]$$

The calculation of G is exactly the above expectation. ∎

Let $X^1, X^2, \ldots, X^r$ denote r independent estimations of X. Consider the average estimator $\bar{X} = \frac{1}{r}(X^1 + \ldots + X^r)$. By Chebychev's inequality, the error is the following

$$\mathsf{Pr}\left\{|\bar{X} - (M'_{\alpha C} - |S|)| < \left(\frac{8G_{\alpha,C}}{r}\right)^{1/2}\right\} > \frac{7}{8}$$

Thus, the error is $O((G_{\alpha,C}/r)^{1/2})$.

$$G_{\alpha,C} \leq \sum_{[a,b] \in \textit{dense}_{s,\alpha C}} \frac{m}{C - (b' - a)} (\#\text{ segments } [a_1, b_1] \text{ s.t. } a_1 \leq a \text{ and } b'_1 \geq a - C + 1)$$

For a fixed segment $[a, b] \in \textit{dense}_{s,\alpha C}$, the number of segments $[a_1, b_1]$ s.t. $a_1 \leq a$ and $b'_1 \geq a - C + 1$ is at most $C - 1$. The upper bound is attained when each segment is of size 1 in the interval $a - C + 1$ until a. Therefore,

$$\begin{aligned} G_{\alpha,C} &\leq \sum_{[a,b] \in \textit{dense}_{s,\alpha C}} \frac{m}{C - (b' - a)}(C - 1) \\ &\leq \sum_{[a,b] \in \textit{dense}_{s,\alpha C}} \frac{m}{(1 - \alpha)C}(C - 1) \\ &= \frac{m(C - 1)}{(1 - \alpha)C}(M'_{\alpha C} - |S|) < mM'_{\alpha C}/(1 - \alpha) \ . \end{aligned}$$

Relative error is given by $\epsilon = \frac{1}{M'_{\alpha C} - |S|}\left(\frac{8G_{\alpha,C}}{r}\right)^{1/2}$

$$< \left(\frac{8mM'_{\alpha C}}{r(M'_{\alpha C} - |S|)^2(1 - \alpha)}\right)^{1/2} \approx O\left(\frac{\sqrt{m}}{\sqrt{M'_{\alpha C} r(1 - \alpha)}}\right)$$

Space requirement for ϵ-accuracy is

$$O\left(\frac{mC}{\epsilon^2 M'^2_{\alpha C}(1 - \alpha)}\right), \quad 0 < \alpha < 1 \ .$$

V. Experimental Validation

In this section, we present detailed experimental results that validate our cache miss estimation technique across a number of cache organizations with varying capacity, associativity, and block size. We measure the relative error of our technique against full cache simulation of a number of important micro-kernels and dominant loops drawn from dense linear algebra and the SPEC95 benchmark suite. In the following, we show the results for matrix-matrix multiplication, a five-pointed system, 3-D stencil, the loop kernel 140 of `101.tomcatv` (from SPEC95), and the nested loop of depth three in `PSINV` subroutine of `107.mgrid` (from SPEC95). Since our technique relies on known loop bounds, in `107.mgrid` we used 2000 as the value of `N`, which is passed to `PSINV` as a parameter. We have created our own cache measurement infrastrucutre to measure the cache misses. This was necessary to observe the

necessary behaviour of interior misses over arrays employed by our ICS method.

Along with the relative error of our technique, we also present the relative errors of four major sampling techniques namely *All References [5],Primed Method [12], Half Warmup Method [8]* and *Stitch Method [1]* have been proposed in the past with the similar goal of reducing the time overhead of the estimation process. All these four techniques first generate a full cache access trace and then sample random clusters from this trace. For more information, the readers should refer to Uhlig and Mudge [15] and Yi et al. [22].

We also show how our technique can be successfully applied to identify the optimal loop permutation in a loop nest (leading to the generation of the corresponding unimodular transformation) and to quickly determine the optimal tile size in a tile-transformed loop. We close this section with a discussion on the speedup achieved by our estimation technique compared to full cache simulation.

A. Relative Estimation Errors on Loop Kernels

We start the discussion of the results by presenting the relative errors of our estimation technique and the other four methods discussed above for the different loop nests running on a 4-way set associative cache with block size of 32 bytes. We consider three cache sizes, namely, 16 KB, 512 KB, and 1 MB. In all the experiments, the number of sampled clusters (n_u) is kept fixed at 300.

1) Matrix-matrix Multiplication: A matrix of size 4096×4096 is chosen for this experiment. Figure 2 shows the relative errors of our ICS method and the four previously proposed schemes. Each plot shows how the relative error varies with the cluster size for a particular cache size. For 16 KB cache, all methods except the *Primed* method deliver error rate below 1% for cluster size higher than 1000. Interestingly, as the cache size increases, only the *Primed* method seems to perform better. However, with a significant increase in cluster size, relative error rate of ICS decreases to about 0.1%. A point worth noting is that the ICS method tends to perform better across all three cache sizes compared to any other single method.

2) Five-pointed System: Five-pointed systems are very popular in iterative solvers where a weighted average of the neighbors of a point is taken as the new value of the point at the end of the current iteration. We show a representative five-pointed kernel below. We use 40 thousand as the value of N in our measurements.

The evaluation across different schemes is presented in Figure 3. Our ICS technique offers error rates below 0.1%, and as already pointed out, the cluster sizes required to achieve this error rate increase significantly in large caches. However, for 1 MB Cache, all techniques except ICS have a significantly high error rate of about 100% even for larger cluster sizes.

3) 3-D Stencil: 3-D stencils are often used in applications like fluid dynamics, heat transfer, etc. This experiment is carried out on the loop structure described in Veldhuizen [16]. We use $N = 4000$. Figure 4 shows the relative errors in estimation.

The relative errors of ICS and *Half warmup* are around 0.1% to 1% for 16 KB cache. As cache size increases to 512 KB and 1 MB, the ICS approach converges to less than 0.1% error for cluster sizes above 35000. A fact worth noting is that none of the other techniques have such low error rates for higher cache sizes.

4) Tomcatv: The estimation results for the selected loop of `tomcatv` are shown in Figure 5. All the techniques tend to stabilize around 1% error rate as the cluster size is increased.

5) Mgrid: Figure 6 shows the relative error rates of different estimation schemes for the selected loop of `mgrid`. For all cache sizes the *All references* method delivers a 100% error rate. Since `mgrid` has a very large working set, this method ends up over-estimating due to scaled up cold misses. For *Primed* method, the situation is reverse, it underestimate because of lesser full sets. Our ICS method tends to stabilize around 1% error as we increase the cache size.*Half warmup* and *Stitch* also offer similar error rates.

B. Effect of Associativity and Block Size

Figure 7 shows the relative error rates of our ICS technique on the 3-D stencil kernel as the associativity is varied from direct-mapped to 8-way. The cache block size is kept fixed at 32 bytes. The results reveal that error rates seems to increase as we move from direct-mapped to 8-way. However, the convergence pattern remains the same due to the nature of references.

Next, we explore the impact of varying the cache block size. Figure 8 shows the relative errors of our ICS technique on the matrix-matrix multiplication kernel for a 512 KB 4-way set associative cache and on the selected loop kernel of `tomcatv` for a 512 KB direct-mapped cache as the cache block size is varied. We find that while increasing the block size from 32 bytes to 128 bytes lowers the estimation error for matrix-matrix multiplication, there is almost no effect of block size on the estimation error in `tomcatv`. The reduction in estimation error with increasing block size for regular kernels like matrix-matrix multiplication can be explained by the fact that increasing block size decreases the number of cache sets. As a result, the number of cold misses decreases leading to better estimation.

C. Application to Loop Optimization

In this section, we show that our ICS technique can be applied to two different kinds of loop optimization. In the first experiment, we use our technique to decide that the `ikj` permutation of the matrix-matrix multiplication loop nest offers the best cache performance. In the second experiment, we show that our technique can correctly decide the optimal tile size for a tile-transformed matrix-matrix multiplication kernel. Both the experiments are carried out for 4096×4096 matrices on a 512 KB 4-way set associative cache with 32-byte block size. Figure 9 presents these results. For the loop permutation experiment, we plot the number of cache misses estimated by our technique for each of the six permutations normalized to the `ijk` permutation as a function of the cluster size. It is very encouraging to note that our technique correctly

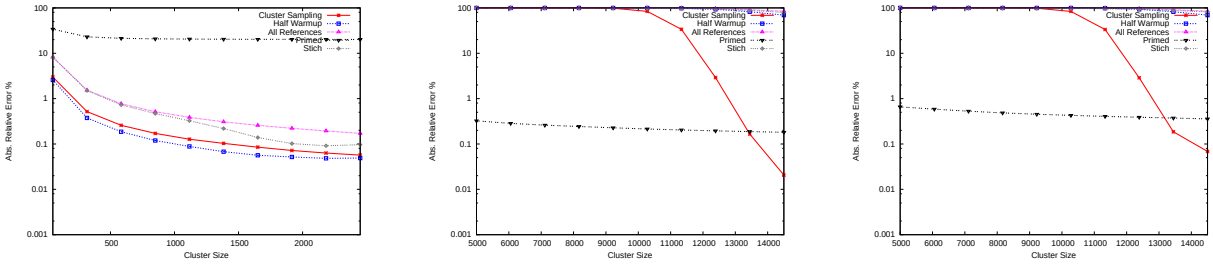


Fig. 2. Comparison of different estimation schemes for a 4096×4096 matrix-matrix multiplication on a 4-way set associative cache with capacity (a) 16 KB, (b) 512 KB, (c) 1 MB.

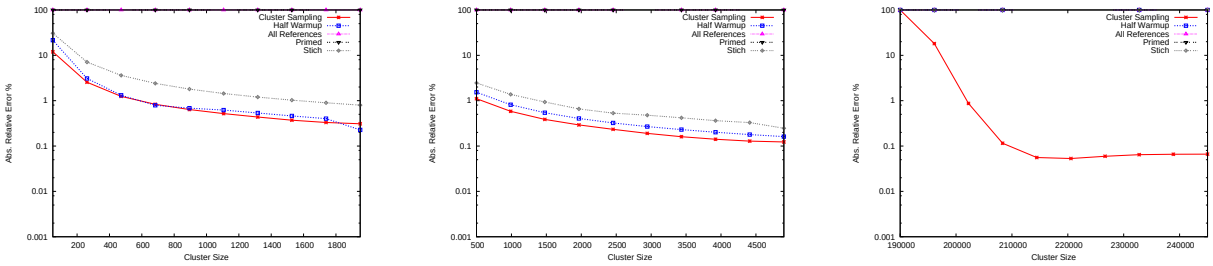


Fig. 3. Comparison of different estimation schemes for a five-pointed system running on a 4-way set associative cache of capacity (a) 16 KB, (b) 512 KB (c) 1 MB.

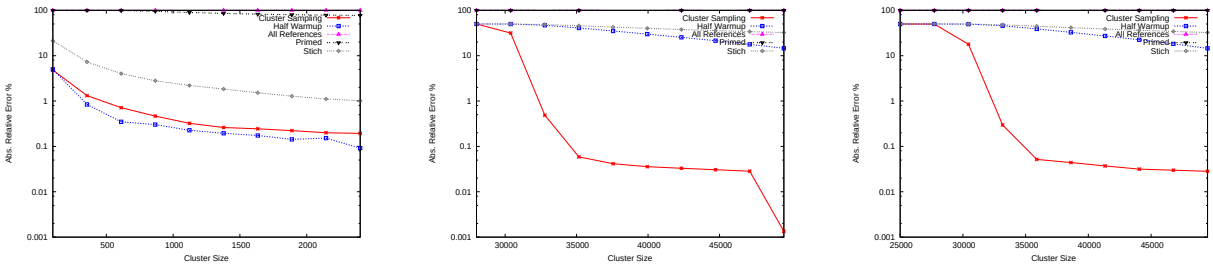


Fig. 4. Comparison of different estimation schemes for a 3-D stencil running on a 4-way set associative cache of capacity (a) 16 KB, (b) 512 KB, (c) 1 MB.

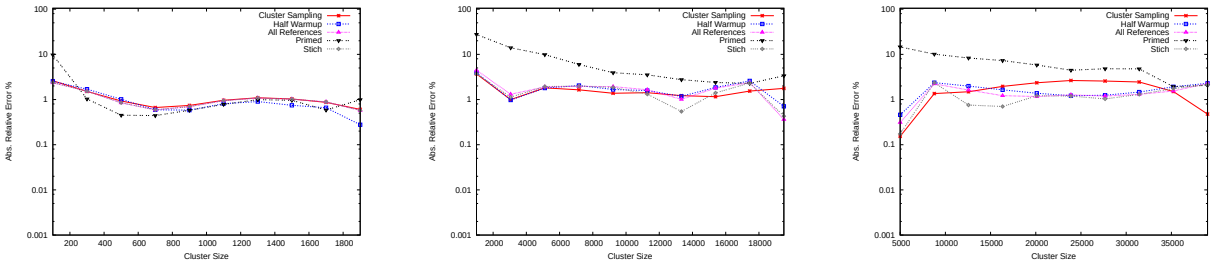


Fig. 5. Comparison of different estimation schemes for the loop kernel 140 of `tomcatv` running on a 4-way set associative cache of capacity (a) 16 KB, (b) 512 KB, (c) 1 MB.

identifies the `jki` and `kji` permutations to be much worse than the `ijk` permutation. It also correctly ranks the `jik` permutation to be worse than the `ijk` permutation. In the figure, `ikj` and `kij` have a small factor of difference, with former being better. Finally, `ikj` permutation is best ranked amongst other permutations. For the tile size experiment, we show the estimated number of cache misses for tile sizes 4, 8, 16, 32, 128, and 256 normalized to no tiling. For the given cache organization, one can verify that, indeed, the 8×8 tile is the optimal one. Our ICS technique correctly identifies this optimal size. It is important to note that a full cache simulation to decide the optimal tile size would take much longer compared to our sampling algorithm. It is not at all difficult to envision a compiler pass that not only tiles a loop nest, but also runs our sampling algorithm to generate the tiled loop with the optimal tile size embedded.

D. Estimation Speedup

Finally, before closing this section, we present the speedup achieved by our estimation scheme compared to full cache

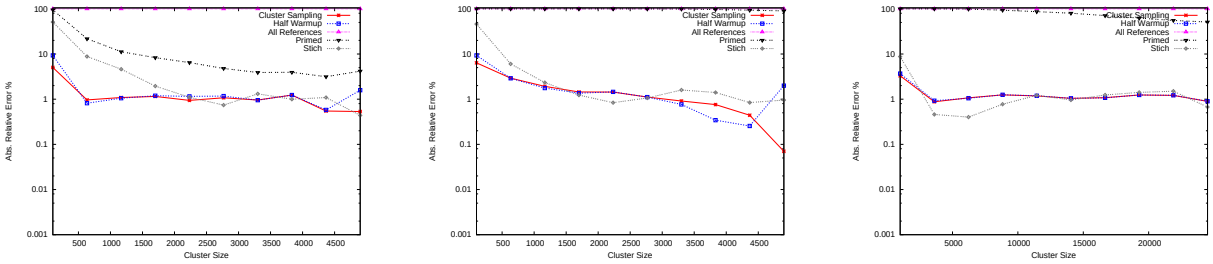


Fig. 6. Comparison of different estimation schemes for the loop kernel in `PSINV` subroutine of `mgrid` running on a 4-way set associative cache of capacity (a) 16 KB, (b) 512 KB, (c) 1 MB.

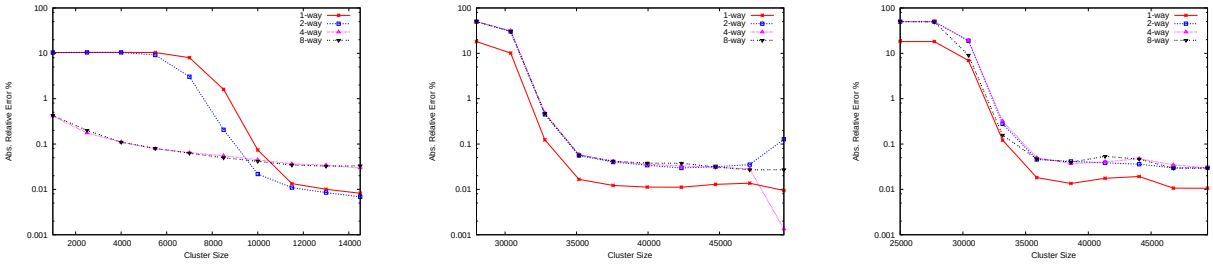


Fig. 7. Effect of associativity on estimation error of ICS for the 3-D stencil kernel with cache capacity of (a) 16 KB, (b) 512 KB, (c) 1 MB.

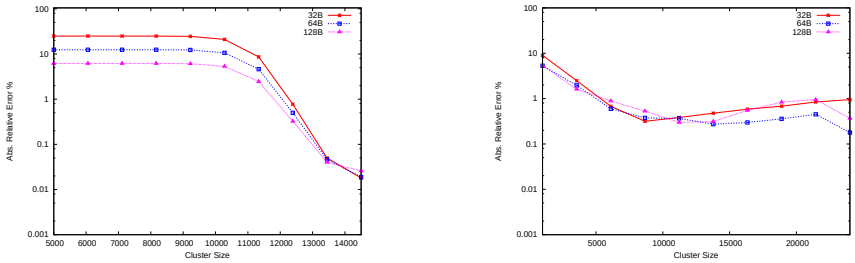


Fig. 8. Effect of cache block size on estimation error of ICS on (a) matrix-matrix multiplication for a 512 KB 4-way set associative cache, (b) `tomcatv` for a 512 KB direct-mapped cache.

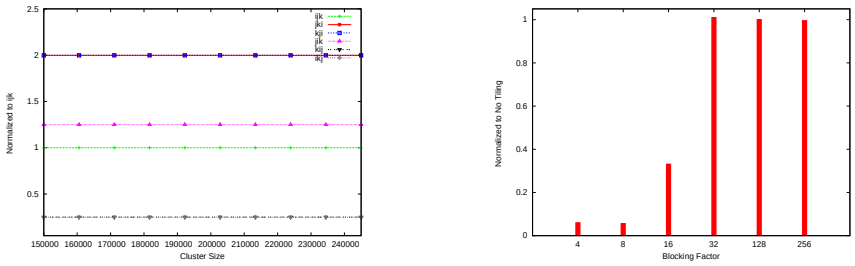


Fig. 9. (a) Estimated number of cache misses in the matrix-matrix multiplication kernel for all six loop permutations normalized to the `ijk` permutation. (b) Estimated number of cache misses for different tile sizes in the tiled matrix-matrix multiplication kernel normalized to no tiling.

simulation. Figure 10 shows these speedup results for all the loop kernels that we have considered (except for the random reference stream) running on a 512 KB cache with 32-byte block size. We present the results for direct-mapped and set associative caches with up to eight ways. In these experiments, the cluster size was chosen such that the estimation error is less than 5%. All these experiments are run on a 3.4 GHz Pentium 4 with 1 GB RAM. The results show that for complex nested loops (e.g., all the loops except five-pointed system and matrix-vector multiplication) our algorithm achieves speedup factors of more than hundred for all associativities. The speedup on 3-D stencil was in the order of thousand. Finally, as expected, the speedup decreases with increasing associativity, since to achieve an error rate of less than 5%, our sampling technique needs to consider bigger cluster sizes (this trend is already discussed above). Overall, we found that our technique always produces results in minutes. These results clearly underscore the feasibility of integrating our ICS scheme in a compiler pass for estimating cache misses of loop nests with known bounds and known array sizes, and using this

estimation to compare the quality of different optimizations.

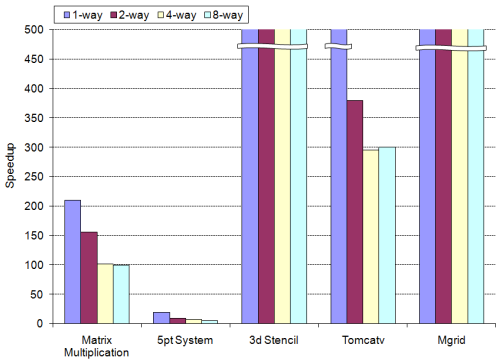


Fig. 10. Speedup of ICS technique compared to full cache simulation on a 512 KB cache with 32-byte block size and different associativities.

VI. Summary and Possible Extensions

This paper has introduced source-level independent cluster sampling as an effective means to estimate the number of cache misses in perfectly nested loops with dense array accesses. Across a number of cache organizations, this technique offers excellent estimates for a number of popular loop kernels drawn from dense linear algebra and the SPEC95 suite. Although in some instances this technique is inferior to some of the previously proposed cache access trace sampling techniques, it offers less than 5% error with more than a factor of hundred speedup in the estimation time compared to full access tracing schemes. For some of the complex loops, the speedup is in the range of thousand. This fast cache miss estimation scheme naturally lends itself to compile-time loop optimization techniques and we successfully demonstrate its applicability to two such optimizations, namely, loop permutation (a popular unimodular transformation) and tile size determination (which is often determined by full execution of loop kernels).

The next natural step would be to integrate this algorithm in a full-fledged compiler pass and apply it to more optimizations related to memory hierarchy. One significant weakness of this technique is that it fails to handle phase behaviors in a loop kernel. Since the sampling technique is not guided by any information drawn from the loop's structure, it would not be surprising if the algorithm misses out on important samples characterizing certain phases of execution. However, it is encouraging to note that the compiler can offer a significant amount of information and feedback about the dynamic regions (i.e., regions in the iteration space) of the loop where the sampling effort should be focused.

VII. Acknowledgements

We would like to thank Nitin Gorde for participating in discussions during the course of this research work.

References

[1] Anant Agarwal, John Hennessy, and Mark Horowitz. Cache performance of operating system and multiprogramming workloads. *ACM Trans. Comput. Syst.*, 1988.

[2] Jon Louis Bentley and James B. Saxe. Generating sorted lists of random numbers. *ACM Trans. Math. Softw.*, 1980.

[3] Calin Cascaval and David A. Padua. Estimating cache misses and locality using stack distances. In *Proceedings of the 17th annual international conference on Supercomputing*, New York, NY, USA, 2003. ACM Press.

[4] Siddhartha Chatterjee, Erin Parker, Philip J. Hanlon, and Alvin R. Lebeck. Exact analysis of the cache behavior of nested loops. In *PLDI '01: Proceedings of the conference on Programming language design and implementation*, New York, NY, USA, 2001. ACM Press.

[5] M.C. Easton. Computation of cold-start miss ratios. *IEEE Transactions on Computers*, May 1978.

[6] Lieven Eeckhout, Smaïl Niar, and Koen De Bosschere. Optimal sample length for efficient cache simulation. *J. Syst. Archit.*, 2005.

[7] Basilio B. Fraguela, Ramón Doallo, and Emilio L. Zapata. Probabilistic miss equations: Evaluating memory hierarchy performance. *IEEE Trans. Comput.*, 2003.

[8] J.W.C. Fu and J.H. Patel. Trace driven simulation using sampled traces. *Twenty-Seventh Hawaii Internation Conference on System Sciences*, 4-7 Jan 1994.

[9] Somnath Ghosh, Margaret Martonosi, and Sharad Malik. Cache miss equations: a compiler framework for analyzing and tuning memory behavior. *ACM Trans. Program. Lang. Syst.*, 1999.

[10] Jr. John W. Haskins and Kevin Skadron. Accelerated warmup for sampled microarchitecture simulation. *ACM Trans. Archit. Code Optim.*, 2005.

[11] Richard Eugene Kessler. *Analysis of multi-megabyte secondary CPU cache memories*. PhD thesis, Madison, WI, USA, 1991.

[12] S. Laha, J. H. Patel, and R. K. Iyer. Accurate low-cost methods for performance evaluation of cache memory systems. *IEEE Trans. Comput.*, 1988.

[13] Erin Parker. *Analyzing the Behavior of Loop Nests in the Memory Hierarchy: Methods, Tools, and Applications*. PhD thesis, North Carolina, USA, 2004.

[14] Erez Perelman, Greg Hamerly, Michael Van Biesbrouck, Timothy Sherwood, and Brad Calder. Using simpoint for accurate and efficient simulation. In *International conference on Measurement and modeling of computer systems*, New York, NY, USA, 2003. ACM.

[15] Richard A. Uhlig and Trevor N. Mudge. Trace-driven memory simulation: a survey. *ACM Comput. Surv.*, 1997.

[16] Todd Veldhuizen. Scientific computing: C++ vs. fortran. *Dr. Dobb's Journal*, 1997.

[17] Xavier Vera and Jingling Xue. Let's study whole-program cache behaviour analytically. In *8th International Symposium on High-Performance Computer Architecture*, page 175, Washington, DC, USA, 2002. IEEE Computer Society.

[18] Xavier Vera, Nerina Bermudo, Josep Llosa, and Antonio González. A fast and accurate framework to analyze and optimize cache memory behavior. *ACM Trans. Program. Lang. Syst.*, 2004.

[19] T.F. Wenisch, R.E. Wunderlich, B. Falsafi, and J.C. Hoe. Simulation sampling with live-points. *International Symposium on Performance Analysis of Systems and Software*, 19-21 March 2006.

[20] David A. Wood, Mark D. Hill, and R. E. Kessler. A model for estimating trace-sample miss ratios. *SIGMETRICS Perform. Eval. Rev.*, 1991.

[21] R.E. Wunderlich, B. Wenisch, T.F.and Falsafi, and J.C. Hoe. Smarts: accelerating microarchitecture simulation via rigorous statistical sampling. *30th Annual International Symposium on Computer Architecture*, 9-11 June 2003.

[22] Joshua J. Yi, Sreekumar V. Kodakara, Resit Sendag, David J. Lilja, and Douglas M. Hawkins. Characterizing and comparing prevailing simulation techniques. In *11th International Symposium on High-Performance Computer Architecture*, Washington, DC, USA, 2005. IEEE Computer Society.

List of Authors

Financial support

The Shared Research Group (SRG) 16-1 on *New Frontiers in High Performance Computing Exploiting Multicore and Coprocessor Technology* is a joint initiative of Karlsruhe Institute of Technology and Hewlett-Packard. The SRG receives grants by the Concept for the Future of Karlsruhe Institute of Technology in the framework of the German Excellence Initiative and by the industrial collaboration partner Hewlett-Packard. The present proceedings of the First International Workshop on New Frontiers in High-performance and Hardware-aware computing are kindly sponsored by the SRG.

High-performance system architectures are increasingly exploiting heterogeneity: multi- and manycore-based systems are complemented by coprocessors, accelerators, and reconfigurable units, providing huge computational power. However, applications of scientific interest (e.g. in high-performance computing and numerical simulation) are not yet ready to exploit the available high computing potential. Different programming models, non-adjusted interfaces, and bandwidth bottlenecks complicate holistic programming approaches for heterogeneous architectures. In modern microprocessors, hierarchical memory layouts and complex logics obscure predictability of memory transfers or performance estimations.

The HipHaC workshop aims at combining new aspects of parallel, heterogeneous, and reconfigurable microprocessor technologies with concepts of high-performance computing and, particularly, numerical solution methods. Compute- and memory-intensive applications can only benefit from the full hardware potential if all features on all levels are taken into account in a holistic approach.

ISBN: 978-3-86644-298-6

www.uvka.de