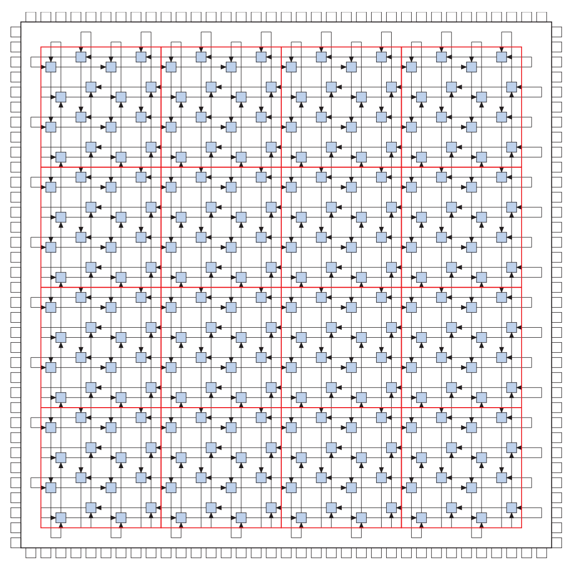
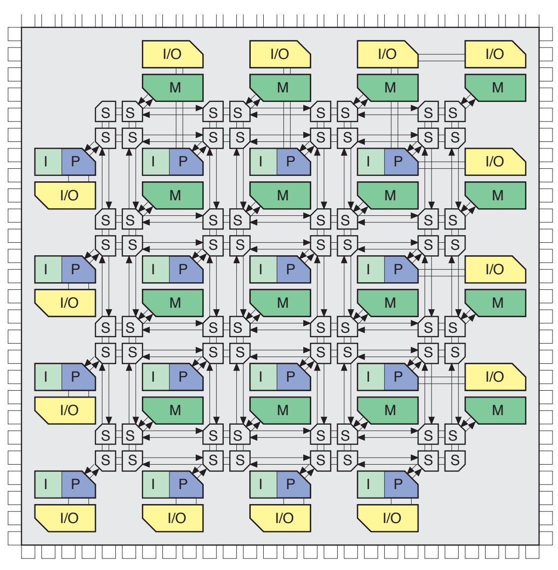




HPPC 2011

(Hand-out) Proceedings of the

5th Workshop on Highly Parallel Processing on a Chip

August 30, 2011, Bordeaux, France
Organizers Martti Forsell and Jesper Larsson Träff

in conjunction with

the 17th International European Conference on Parallel and Distributed Computing (Euro-Par)
August 29-September 2, 2011, Bordeaux, France

Sponsored by

(Hand-out) Proceedings of the

5th Workshop on Highly Parallel Processing on a Chip

August 30, 2011, Bordeaux, France
http://www.hppc-workshop.org/

in conjunction with

the 17th International European Conference on Parallel and Distributed Computing (Euro-Par)
August 29-September 2, 2011, Bordeaux, France

August 2011
Handout editors: Martti Forsell and Jesper Larsson Träff
Printed in Finland and Austria

CONTENTS

Foreword 4

Organization 5

Program 6

SESSION 1 - High throughput computing CMPs

Keynote - Extreme Thread-Level-Parallelism on Sparc Processors - Rick Hetherington, Microelectronics, Oracle Corp 7

Thermal Management of a Many-Core Processor under Fine-Grained Parallelism - Fuat Keceli, Tali Moreshet and Uzi Vishkin, University of Maryland, Swarthmore College 8

SESSION 2 - Programming and optimization of CMPs

Mainstream Parallel Array Programming on Cell - Paul Keir, Paul Cockshott and Andrew Richards, University of Glasgow, Codeplay Software Ltd 18

Generating GPU Code from a High-level Representation for Image Processing Kernels - Richard Membarth, Anton Lokhmotov and Jürgen Teich, University of Erlangen-Nuremberg, ARM 28

A Greedy Heuristic Approximation Scheduling Algorithm for 3D Multicore Processors - Thomas Canhao Xu, Pasi Liljeberg and Hannu Tenhunen, Turku Center for Computer Science, University of Turku 38

FOREWORD

The arrival of multicore processors and plans to increase the amount of cores per chip exponentially over the coming years have made parallel computing reality for the programmers. Unfortunately, usability of available development tools, the asynchronous models of computation used in current designs, heterogeneity of them, and lack of proper education are making parallel programming still very challening. Also the development of multicore architectures is its infancy, adopting still many non-scalable techniques from the sequential computing domain although there would be more efficient parallel computing-aware solutions available. Finally, the technological challenges, including ever increasing mask costs, design complexity explosion of especially heterogeneous chips, inclusion of third dimension, power density concerns, and raising influence of atom-level effects are threatening to diminish returns form silicon process improvements. Thus, there is a strong need to study, develop and teach parallel languages, compiling, models of computation, architctural techniques, advanced implementation technologies, and entire execution architectures before parallel applications will be efficiently executed on highly parallel multicore processors and parallel computing becomes the main stream of computing.

This is fifth time we organize the Workshop on Highly Parallel Processing on a Chip (HPPC). Again, it aims to be a forum for discussing such fundamental issues. It is open to all aspects of existing and emerging/envisaged multi-core processors with a significant amount of parallelism, especially to considerations on novel paradigms and models and the related architectural and language support. To be able to relate to the parallel processing community at large, which we consider essential, the workshop has been organized in conjunction with Euro-Par, the main European (and international) conference on all aspects of parallel processing.

The Call-for-papers for the HPPC workshop was launched on March, and at the passing of the submission deadline we had received 7 submissions, which were relevant to the theme of the workshop and of good quality. The papers were swiftly and expertly reviewed by the program committee, all of them receiving 5 qualified reviews. We thank the whole of the program committee for the time and expertise they put into the reviewing work, and for getting it all done within the rather strict timelimit. Final decision on acceptance was made by the program chairs based on the recommendations from the program committee. This year the themes of manuscripts matched well to the scope of the workshop and we were able to accept full 4 contributions, resulting in an acceptance ratio of about 57%. The 4 accepted contributions will be presented at the workshop today, together with a forward looking invited talk by Rick Hetherington on extreme thread-level parallelism on Oracle Sparc processors.

This handout includes the workshop versions of the HPPC papers and the abstracts of the invited talks. Final versions of the papers will be published as post proceedings in a Springer LNCS volume containing material from all the Euro-Par workshops. We sincerely thank the Euro-Par organization for giving us the opportunity to arrange the HPPC workshop in conjunction with the Euro-Par 2011 conference. We also warmly thank our sponsors VTT, University of Vienna and Euro-Par for the financial support which made it possible for us to invite Rick Hetherington, of whom we also sincerely thank for accepting our invitation to come and contribute.

Finally, we welcome all of our attendees to the Workshop on Highly Parallel Processing on a Chip in the beautiful city of Bordeaux, France. We wish you all a productive and pleasant workshop.

HPPC organizers
Martti Forsell, VTT, Finland
Jesper Larsson Träff, University of Vienna, Austria

ORGANIZATION

Organized in conjuction with the 17th International European Conference on Parallel and Distributed Computing

WORKSHOP ORGANIZERS

Martti Forsell, VTT, Finland
Jesper Larsson Träff, University of Vienna, Austria

PROGRAM COMMITTEE

David Bader, Georgia Institute of Technology, USA
Martti Forsell, VTT, Finland
Jim Held, Intel, USA
Peter Hofstee, IBM, USA
Magnus Jahre, NTNU, Norway
Chris Jesshope, University of Amsterdam, The Netherlands
Ben Juurlink, Technical University of Berlin, Germany
Jörg Keller, University of Hagen, Germany
Christoph Kessler, University of Linköping, Sweden
Avi Mendelson, Microsoft, Israel
Vitaly Osipov, Karlsruhe Institute of Technology, Germany
Martti Penttonen, University of Eastern Finland, Finland
Sven-Bodo Scholz, University of Hertfordshire, UK
Jesper Larsson Träff, University of Vienna, Austria
Theo Ungerer, University of Augsburg, Germany
Uzi Vishkin, University of Maryland, USA

SPONSORS

VTT, Finland	http://www.vtt.fi
University of Vienna	http://www.univie.ac.at
Euro-Par	http://www.euro-par.org

PROGRAM

5th Workshop on Highly Parallel Processing on a Chip (HPPC 2011)

TUESDAY AUGUST 30, 2011 Bordeaux

SESSION 1 - High throughput computing CMPs

09:30-09:35 Opening remarks - *Jesper Larsson Träff and Martti Forsell, University of Vienna, VTT*
09:35-10:35 Keynote - Extreme Thread-Level-Parallelism on Sparc Processors - *Rick Hetherington, Microelectronics, Oracle Corp, USA*
10:35-11:00 Thermal Management of a Many-Core Processor under Fine-Grained Parallelism - *Fuat Keceli, Tali Moreshet and Uzi Vishkin, University of Maryland, Swarthmore College*

11:00-11:30 -- Break --

SESSION 2 - Programming and optimization of CMPs

11:30-11:55 Mainstream Parallel Array Programming on Cell - *Paul Keir, Paul Cockshott and Andrew Richards, University of Glasgow, Codeplay Software Ltd*
11:55-12:20 Generating GPU Code from a High-level Representation for Image Processing Kernels - *Richard Membarth, Anton Lokhmotov and Jürgen Teich, University of Erlangen-Nuremberg, ARM*
12:20-12:45 A Greedy Heuristic Approximation Scheduling Algorithm for 3D Multicore Processors - *Thomas Canhao Xu, Pasi Liljeberg and Hannu Tenhunen, Turku Center for Computer Science, University of Turku*
12:45-12:50 Closing remarks - *Jesper Larsson Träff and Martti Forsell, University of Vienna, VTT*

12:50-14:30 -- Lunch --

14:30-15:00 Informal business meeting on the Highly Parallel Processing on a Chip Workshop Series - *Jesper Larsson Träff and Martti Forsell, University of Vienna, VTT*

KEYNOTE

Extreme Thread-Level-Parallelism on Sparc Processors

Rick Hetherington, Vice President, Microelectronics, Oracle Corp, USA

Abstract: Sparc has been the leader in pursuing high levels of throughput on commercial workloads with the use of Chip-Multithreaded Processors. Introduced in 2005, the Niagara 1 processor broke new ground by providing 32 processor threads across 8 cores. This talk will revisit the history of Sparc CMT as well as presenting the current state of CMT with the fourth generation of Sparc CMT and some projections as to where this technology is heading in future. The talk will touch on experience gained, lessons learned, what worked and what did not work during this nearly decade of experience on highly threaded processor design.

Bio: *Rick Hetherington is Vice President in the Microelectronics Division of Oracle. He is responsible for Sparc Architecture and Performance. In this role, he manages a team of architects and performance analysts to develop processors and systems.*

Rick Hetherington was the chief technology officer for Sun's Microelectronics business unit where he set the technical direction for Sun's SPARC processor development and related technologies. Hetherington is driving Sun's leadership in the chip multithreading (CMT) approach to processor design, in which multiple cores and multiple threads combine to generate extraordinary throughput and power efficiency.

Prior to his appointment as CTO, Hetherington provided oversight for the architecture, performance and roadmap of UltraSPARC T1(TM) processors and systems as chief architect for Sun's Horizontal Systems group. From 2000 to 2002, Hetherington took a hiatus from Sun to join a networking start-up as VP of engineering.

Hetherington originally joined Sun in 1996 as co-architect of Sun's Project Millennium processor. Prior to Sun, he spent 16 years with Digital Equipment Corp., working on a variety of VAX and Alpha processors and systems. His last position at Digital was system architect of EV6 (21264) processor.

Hetherington earned his bachelor's degree from Pennsylvania State University and currently holds 63 patents.

Thermal Management of a Many-Core Processor under Fine-Grained Parallelism

Fuat Keceli[1], Tali Moreshet[2], Uzi Vishkin[1]

[1]University of Maryland, College Park, MD, USA
[2]Swarthmore College, Swarthmore, PA, USA

Abstract. In this paper, we present the work in progress that studies the run-time impact of various DTM techniques on a proposed 1024-core XMT chip. XMT aims to improve single task performance using fine-grained parallelism. Via simulations, we show that relative to a general global scheme, speedups of up to 46% with a dedicated interconnection controller and 22% with distributed control of computing clusters are possible. Our findings lead to several high level insights that can impact the design of a broader family of shared memory many-core systems.

1 Introduction

Thermal feasibility has become a first-class architectural design constraint reflected in specifications of modern processors. It is typical to incorporate dynamic thermal management (DTM) in order to improve the power envelope without having to adopt more expensive cooling solutions. Going forward, it is important to advance the understanding of DTM techniques for efficiently supporting the architecture trend of increase in core count.

We base our work on the eXplicit Multi-Threading (XMT) architecture. XMT is a general-purpose many-core platform for fine-grained parallel programs, with significant evidence on ease-of-programming (e.g., [24]) and competitive performance (e.g., [3]). The reasons for choosing XMT as our platform are: *(a)* As noted in [25], ease-of-programming is crucial for the success of parallel computers and XMT constitutes a competitive and realistic direction in this respect, *(b)* The XMT simulator allows for evaluation of DTM techniques and such an infrastructure is not publicly available for other current many-core platforms.

In this paper, we evaluate the potential benefits of several DTM techniques on XMT. We focus on a system which executes one parallel task at a time. The relevance and the novelty of our work can be better understood by answering the following two questions.

Why is single task fine-grained parallelism important? On a general-purpose many-core system the number of concurrent tasks is unlikely to often reach the number of cores (i.e., thousands). Parallelizing the most time consuming tasks is a sensible way for both improved performance and taking advantage of the plurality of cores. The main obstacle then is the difficulty of programming for single-task parallelism. Scalable fine-grained parallelism is natural for easy-to-program approaches such as XMT.

What is new in many-core DTM? DTM on current multi-cores mainly capitalizes on the fact that cores show different activity levels under multi-tasked workloads [5]. In a single-tasked many-core, the source of imbalance is likely to

lie in the structures that did not exist in the former architectures such as the large scale on-chip interconnection network (ICN) and distributed shared caches.

Contribution. This paper introduces XMTSim+dtm, a new, DTM-enabled cycle-accurate simulator based on XMTSim [16]. Using XMTSim+dtm we measure the performance improvements introduced by several DTM techniques for a 1024-core XMT chip. We compare techniques that are tailored for a many-core architecture against a global DTM (GDVFS), which is not specific to many-cores. Following are the highlights of the insights we provide: (a) Workloads with scattered irregular memory accesses benefit more from a dedicated ICN controller (up to 46% runtime improvement over GDVFS). (b) In XMT, cores are arranged in clusters. Distributed DTM decisions at the clusters provide up to 22% improvement over GDVFS for high-computation parallel programs, yet overall performance may not justify the implementation overhead.

Our work is relevant for architectures that consider similar design choices as XMT (for example the Plural system [7]) which promote the ability to handle both regular and irregular parallel general-purpose applications competitively (see Section 3.1 for a definition of regular and irregular). These design choices include an integrated serial processor, no caches that are local to parallel cores, and a parallel core design that provides for a true SPMD implementation. We aim to establish high-level guidelines for designers of such systems.

A comprehensive body of previous work is dedicated to dynamic power and thermal management techniques for multi-core processors [13]. However, in most cases (e.g., [6, 12, 21]), authors assume a pool of uncorrelated serial benchmarks as their workload, and capitalize on the variance in the execution profiles of these benchmarks. The study by Ma, et al. [22] is notable, as they simulate a set of parallel benchmarks, however, it focuses on power rather than thermal management and considers up to only 128 cores. To our knowledge, our work is among the first to evaluate DTM techniques on a many-core processor for single task parallelism.

2 Experimental Platform

2.1 The XMT Architecture

The primary goal of the eXplicit Multi-Threading (XMT) general-purpose computer architecture [28] has been improving single-task performance through parallelism. XMT was designed from the ground up to capitalize on the huge on-chip resources becoming available in order to support the formidable body of knowledge, known as Parallel Random Access Model (PRAM) algorithmics [18], and the latent, though not widespread, familiarity with it. Driven by the repeated programming difficulties of parallel machines, ease-of-programming was a leading design objective of XMT. Indeed, considerable amount of evidence was developed on *ease of teaching* and improved *development time* with XMT (e.g., [24])[1].

The XMT architecture includes an array of lightweight cores, Thread Control Units (TCUs), and a serial core with its own cache (Master TCU). The architecture includes several clusters of TCUs connected to mutually-exclusive shared

[1] A complete list of references can be found at `http://www.umiacs.umd.edu/users/vishkin/XMT/index.shtml#publications`

cache modules by a high-throughput interconnection network [1] (XMT does not feature writable private caches). TCUs include lightweight ALUs, but the more heavy-weight units are shared by all TCUs in a cluster. XMT is programmed in XMTC, a simple extension of the C language which contains succession of serial and parallel code sections. The code of a parallel section is expressed in the SPMD (single program, multiple data) style, specifying an arbitrary number of virtual threads sharing the same code. Further details on the XMT architecture can be found in [28].

2.2 Simulation Environment

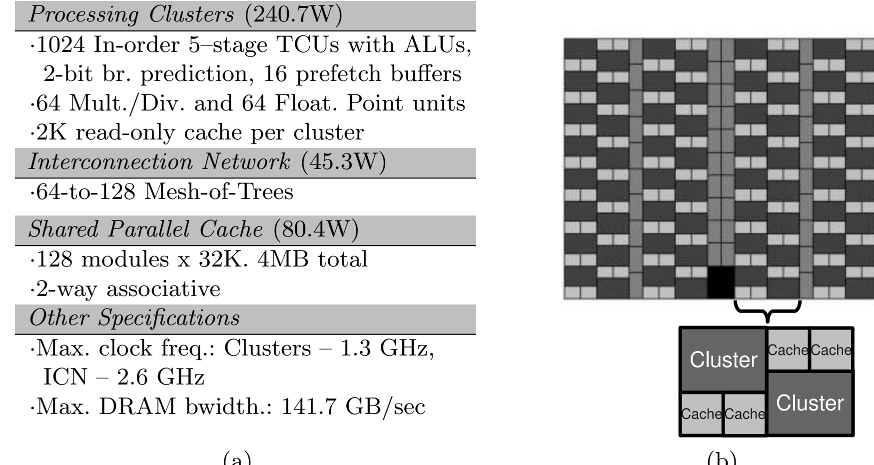
The simulation environment used in this work is XMTSim+dtm, an extension of publicly available XMTSim [16]. XMTSim is the cycle-accurate simulator of the XMT computer architecture, built to model the on-chip components that constitute the ASIC and FPGA prototypes of XMT. XMTSim has been validated against the FPGA prototype and cycle counts are shown to be accurate within a margin of 15% [17]. The parameters of the power model used in XMTSim is based on other validated tools: McPat [20] and Cacti [29]. The details of this estimation are the subject of a companion paper [15]. More details on the simulator than presented hereafter can be found in [14, 16, 17].

The high level specifications of the 1024-TCU XMT chip that we simulated in our experiments are given in Table 1a. The values in the table were compiled to match the chip area of an advanced GPU, NVIDIA GTX280. Details of this analysis are given in [3].

Power Estimation. XMTSim estimates power based on the activity that a benchmark induces on the clusters, ICN and the shared caches. Effect of die temperatures is included in calculation of the leakage power. The power model used in XMTSim is similar to the model proposed in [11] and explained in detail in [14, 17]. The model reflects the assumptions that a components implement clock gating and voltage gating. Table 1a lists the maximum powers of the components in the simulated XMT chip.

The Temperature Model. XMTSim uses HotSpot [27] to estimate the temperatures of the clusters and the maximum temperature of the area dedicated to the ICN routing. We set the thermal limit at 65C, whenever required, and simulated our benchmarks with a heatsink convection resistance of 0.1W/K, observed in typical cooling solutions [10]. Fig. 1b denotes the simulated XMT floorplan. It is motivated by previous work that shows it is thermally more efficient to place the clusters and shared caches in a checkerboard pattern rather than keeping them separately [10].

Simulating DTM. XMTSim samples the power and computes temperature at regular intervals. At each interval, DTM algorithm calculates a new set of frequencies for the clusters and the ICN. In order to avoid very long simulation times, we use steady-state temperature computations. Steady-state is an approximation to transient solutions for very long intervals with steady inputs. We observed that the behaviors of the kernels we simulated do not change significantly with larger data sets, except that the phases of consistent activity stretch in time. Therefore, we interpret the steady-state results obtained from simulating relatively short kernels with narrow sampling intervals as indicators



Processing Clusters (240.7W)
·1024 In-order 5–stage TCUs with ALUs, 2-bit br. prediction, 16 prefetch buffers
·64 Mult./Div. and 64 Float. Point units
·2K read-only cache per cluster
Interconnection Network (45.3W)
·64-to-128 Mesh-of-Trees
Shared Parallel Cache (80.4W)
·128 modules x 32K. 4MB total
·2-way associative
Other Specifications
·Max. clock freq.: Clusters – 1.3 GHz, ICN – 2.6 GHz
·Max. DRAM bwidth.: 141.7 GB/sec

(a)

(b)

Table 1. (a) The specifications of the 1024-TCU XMT. Given in parentheses are the maximum cumulative power for each group of components. (b) Checkerboard floorplan for a 1024 TCU XMT. Vertical strips with light gray color are reserved for interconnection network routing and the black colored rectangle is the Master TCU.

of potential results from longer kernels. The sampling intervals, ranging from 5K to 200K clock cycles, are determined empirically to filter the noise in the activity patterns.

3 Performance Under Thermal Constraints

In this section, we discuss the performance of the XMT chip under thermal constraints and evaluate a set of dynamic thermal management techniques that can potentially improve the performance. Our main objective is obtaining the shortest execution time for a benchmark without exceeding a predetermined temperature limit. Note that energy efficiency is not within the scope of this objective.

3.1 Benchmarks

We consider it essential for a general-purpose architecture to perform well on a range of applications. Therefore we include both regular benchmarks, such as graphics processing, and irregular benchmarks, such as graph algorithms. In a typical regular benchmark, memory access addresses are predictable and there is no variability in control flow. In an irregular benchmark, memory access addresses and the control flow (if it is data dependent) are less predictable. Since it is our purpose to make the results relevant to other many-core platforms, we select benchmarks that commonly appear in the public domain [2, 4, 8, 23, 26]. Where applicable, benchmarks use single precision floating point format, except FFT, which uses fixed point arithmetics (i.e., utilizing the integer functional units instead of the FP units).

Table 2 provides a summary of the benchmarks that we use in our experiments. Execution times (in clock cycles), average power and temperature values are obtained from simulating benchmarks with no thermal constraint. Power and temperature estimations assume a global clock frequency of 1.3 GHz and

Name	Results	Description	Characteristic and Dataset
BFS-I	1.825M, 161W, 60C	Breadth-first search on graphs.	Hi-P. 1M nodes, 6M edges. Low cluster and moderate ICN activity. Irregular.
BFS-II	135.2M, 118W, 56C		Low-P. 200K nodes, 1.2M edges. Very low activity. Irregular.
Bprop	3.990M, 118W, 56C	Back Propagation machine learning alg.	Hi-P. 64K nodes. Low activity. Irregular.
Conv	0.885M, 215W, 66C	Image convolution with separable filter.	Hi-P. 1024x512. Highest activity. Regular.
FFT	4.905M, 194W, 63C	1-D fixed point Fast-Fourier transform.	Hi-P. 1M points. Moderate activity. Regular.
Mmult	10.6M, 180W, 63C	Multiplication of two integer matrices.	Low-P. 512x512 elts. Moderate cluster and high ICN activity. Regular.
Msort	3.625M, 161W, 60C	Merge-sort algorithm.	Hi-P. 1M keys. Variable moderate to low activity. Irregular.
NW	1.725M, 166W, 61C	Needleman-Wunsch sequence alignment.	Low-P. 2x2048 seqs. Variable moderate to low activity. Irregular.
Reduct	0.67M, 185W, 63C	Parallel reduction (sum).	Hi-P. 16M elts. Moderate cluster and high ICN activity. Regular.
Spmv	0.31M, 205W, 64C	Sparse matrix - vector multiplication.	Hi-P. 36Kx36K, 4M non-zero. Moderate cluster and high ICN activity. Irregular.

Table 2. Benchmark properties. Results are clock cycles, average power and temperature, consecutively.

$R_c = 0.05W/K$. Each benchmark is characterized in terms of its degree of parallelism, regularity and activity. More detail on these will follow next.

The degree of parallelism for a benchmark is defined to be low (*low-P*) if the number of TCUs executing threads is significantly smaller than the total number of TCUs when averaged over the execution time of the benchmark. Otherwise the benchmark is categorized as highly parallel (*hi-P*). According to Table 2, three of our benchmarks, *BFS-II*, *Mmult* and *NW*, are identified as low-P. In *Mmult*, the size of the multiplied matrices is 512×512 and each thread handles one row, therefore only half of the 1024 TCUs are utilized. *BFS-II* shows a random distribution of threads between 1 and 11 in each one of its 300K parallel sections. *NW* shows varying amount of parallelism between the iterations of a large number of synchronization steps (i.e., parallel sections in XMTC).

As mentioned above, regularity of a benchmark is affected by the predictability of memory access patterns and the control flow. For instance, *BFS*, *Msort* and *Spmv* are irregular due to their memory access patterns, whereas *BFS* also shows data dependent control flow. *FFT* is another benchmark with irregular memory access patterns, however it is classified as regular since the uniform distribution of work among TCUs was the dominant factor in this case. Some of the other factors that play a role in the regularity of a benchmark are the amount and variability of parallelism (e.g., *NW*), and memory bottlenecks (e.g., *Bprop*). *Bprop* is a complex irregular program with heavy memory queuing.

The activity profile of a benchmark plays an important role in the behavior of the system under thermal constraints, as we demonstrate in Section 3.3. It can be observed in Table 2 that there is a direct correlation between the regularity and activity/power value of a benchmark. For example, *Conv*, *Reduct* and *Mmult* are typical regular benchmarks with steady and high activity and power values.

In addition to regularity, cluster and ICN activities are determined by a number of additional factors, such as the computation to memory operation ratio

of the threads and amount of parallelism (even if it is constant). An example is fixed-point *FFT*, which has lower activity than the other regular benchmarks, partly because it performs integer operations rather than floating point, spending less time in computation. Another example, *Mmult*, despite being very regular, is not as active as *Conv* due to the fact that it is a low-P benchmark.

3.2 Dynamic Thermal Management

Dynamic thermal management is the general term for various algorithms used to increase, or more efficiently utilize, the power envelope without exceeding a limit temperature at any location on the chip, as observed by thermal sensors. DTM alters the operation of the system during runtime via tools such as dynamic voltage and frequency scaling (DVFS), clock gating and voltage gating. These tools are commonly implemented in current processors [9, 13, 19].

In our experiments, we evaluated the following DTM techniques that are motivated by previous work on single and multi-cores [13]. We adapted these techniques to our many-core platform. The DTM decisions for each technique are determined during runtime and based on proportional-integral (PI) controllers which take temperatures as input and output the clock frequency and voltage level [27]. The clusters and ICN are assumed to feature one or more temperature sensors that report their maximum temperatures. The exact sensor configuration is beyond the scope of this work. In the distributed DVFS algorithms, a controller assigned to a certain component in the chip (for example, the ICN or a cluster) responds to the temperature of that component. Further details can be found in [14].

Global DVFS (GDVFS): All clusters, caches and the ICN are connected to one central controller which converges to the maximum frequency/voltage values possible without exceeding limit temperature. Global DVFS is the simplest DTM technique in terms of physical implementation, therefore, any other technique should perform better in order to justify its added design complexity.

Coarse-Grain Distributed DVFS (CG-DDVFS): The ICN is assigned a separate controller while the rest of the chip remains connected to the global controller as in GDVFS. Some many-cores, such as GTX280, already have separate clock domains for the computation elements and the interconnection network, leading us to conclude that the implementation cost of this technique is acceptable.

Fine-Grain Distributed DVFS (FG-DDVFS): Each cluster is connected to an isolated voltage/frequency domain and assigned a separate controller. The shared caches are connected to a common domain and their frequency is set to the average frequency of the clusters. The ICN is assigned a dedicated controller as in CG-DDVFS. The implementation of distributed DVFS may be prohibitively expensive on large systems due to the high number of voltage/frequency domains.

3.3 Analysis of DTM Results

A chip with no dynamic thermal management is designed to work at the highest possible clock frequency with which it can tolerate the thermal stress of the worst case (i.e., the most active, most power intensive) benchmark. Consequently, applications that are not as thermally demanding are penalized, since

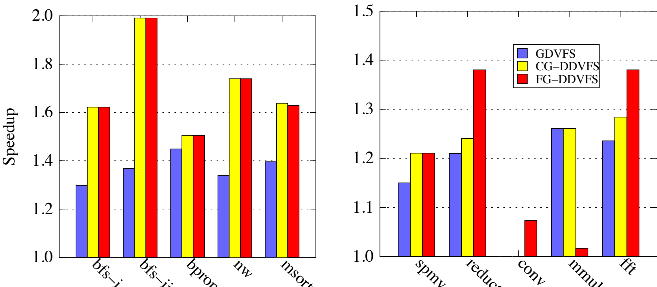


Fig. 1. Benchmark speedups for the DTM algorithms. The benchmarks are grouped into low (left) and high (right) cluster activity. Note that the two groups have different y-axis ranges.

they are subject to the same clock frequency. DTM techniques provide the highest improvement on the execution time of such benchmarks compared to the *no-DTM* case. Consistent with this statement, throughout this analysis it can be observed that benchmarks identified as low activity in Section 3.1 show the highest speedups with DTM techniques.

In evaluating the DTM techniques introduced in Section 3.2, we set an XMT chip with *no-DTM* as the baseline and express the performance of a DTM technique in terms of speedups over the baseline. We assume that the *no-DTM* system is optimized to run at the fastest clock frequency that is thermally feasible for *Conv*, which is the most thermally active benchmark according to Section 3.1. We determined the baseline clock frequency as 900MHz. The thermal limit was set at 65C, as indicated in Section 2.2.

In Fig. 1, we present the benchmark speedups when simulated with the examined DTM techniques. If thermal management is present, the clock frequencies of the clusters and caches can be dynamically scaled up to 1.3GHz. We also assume that ICN frequency can be raised to a maximum of 2.6GHz. A higher interconnect speed is possible due to the simplicity of the pipeline stages in the Mesh-of-Trees ICN (as in [9]). The speedup of a benchmark is calculated using the following formula: $S = Exec_{base}/Exec_{dtm}$, where $Exec_{base}$ and $Exec_{dtm}$ are the execution times on the baseline, *no-DTM* system and with thermal management, respectively.

As a general trend, the benchmarks that benefit the most from the DTM techniques are benchmarks with low cluster activity factors, namely *BFS-I*, *BFS-II*, *Bprop*, *NW* and *Msort* (note the scale difference between the y axes of the two rows of Fig. 1). The highest speedups are observed for *BFS-II* since it has very low overall activity and runs at the maximum clock speed without even nearing the limit temperature. Moreover, CG-DDVFS and FG-DDVFS performs up to 46% better than GDVFS for *BFS-II*, which is mainly bound by memory latency, hence benefits greatly from the independently increased ICN frequency. On the other extreme, *Conv* has the highest cluster activity and was used as the worst case in determining the feasible baseline clock frequency, and therefore shows the least improvement in most experiments.

In the remainder of this section we elaborate on the performance of CG-DVFS and FG-DVFS separately.

CG-DDVFS. Our experiments show that the CG-DDVFS algorithm presents a very reasonable trade-off between hardware complexity and performance. However, performance of certain benchmarks can suffer from the CG-DDVFS algorithm without the modification explained next.

We observed that the performance of the originally proposed CG-DDVFS algorithm is adversely affected by the fact that the ICN and cluster temperatures are correlated. The conduction of the heat generated by the ICN activity adds to the temperature of the clusters, requiring a slowdown in the cluster clock frequency. Moreover, when the ICN frequency is increased, the cores spend less time idling on memory operations and the cluster power escalates. As a consequence, CG-DDVS is most effective on benchmarks with low cluster activity, and may hurt the performance of a computation bound benchmark by causing the cluster frequency to drop excessively while trying to increase the ICN frequency for an insignificant gain. Therefore, we modified the algorithm to fall back to GDVFS when the ICN activity is lower than the cluster activity (this criterion was determined empirically). Fig. 1 reflects the results of this modification.

The CG-DDVFS technique provides better performance than GDVFS on *BFS-I* (25% faster), *BFS-II* (46% faster), *NW* (30% faster) and *Msort* (17% faster), which have irregular memory accesses and low computation to memory operation ratios. For these benchmarks, the performance bottleneck is the amount of time that TCUs wait on memory operations. CG-DDVFS shortens the wait time by increasing the ICN clock frequency. Irregularity of memory accesses implies that the ICN is not saturated, and it is not fully utilizing its power envelope. Therefore, ICN has sufficient thermal slack that can be picked up by increasing its clock frequency. *Bprop* does not benefit from CG-DDVFS significantly, due to the high degree of queuing on the shared memory modules with this benchmark. We also observed that CG-DDVFS can cause a performance degradation of up to 12% with respect to GDVFS for the remainder of the benchmarks if the GDVFS fall-back is not implemented.

Insight: For a system with a central ICN component such as XMT, workloads that are characterized by scattered irregular memory accesses usually benefit more from dedicated ICN thermal monitoring and control. Conversely, CG-DDVFS can hurt the performance and should be disabled for regular parallel programs which usually have higher computation to memory operation ratios. Performance of CG-DDVFS improves for more advanced cooling solutions.

FG-DDVFS. For the single-tasked system such as the one we examine in this work, the activity does not vary significantly among the clusters when averaged over a sufficiently long time window. The said time window is shorter than the duration required for a significant change in temperature to occur. Therefore, the only benefit that FG-DDVFS can provide is due to the temperature gradient between the middle of the die, where it is harder to remove the heat, and the edges. FG-DDVFS tries to pick up the thermal slack at the edge clusters by increasing their clock frequency. However, the performance of FG-DDVFS suffers

from the interaction between the temperatures of the center and edge clusters: as the temperature of the edge clusters rises, so does the temperature of the center clusters, and the controllers in the middle will respond by converging at lower clock frequencies, diminishing the performance gain.

FG-DDVFS exhibits speedups similar to CG-DDVFS on the low activity benchmarks. This is due to the dedicated ICN clock controller introduced in CG-DDVFS. The added value of the dedicated cluster controllers of FG-DDVFS is observed on *Spmv*, *Reduct*, *Conv* and *FFT*, which are the high activity benchmarks. For these benchmarks, runtimes with FG-DDVFS are faster than GDVFS by 14%, 7% and 11%, respectively. *Mmult* is the only benchmark that shows a slowdown over GDVFS due to its low-P characteristic.

Insight: Individual temperature monitoring and control for computing clusters may be worthwhile even in a single-tasking system with fairly uniform workload distribution. The gains are noteworthy for regular parallel programs with high amounts of computation. Conversely, the overall performance of FG-DDVFS on the low activity benchmarks may not justify its added cost for some systems. It should also be noted that FG-DDVFS has advantage over CG-DDVFS only for cases where the overall speedups are lower than average.

4 Conclusion

In this paper, we outline ongoing work in which we tailor various DTM techniques that exist in uniprocessors and multi-cores to a many-core architecture. We evaluate these techniques on the easy-to-program XMT many-core architecture according to their implementation/design complexity and improvement in single task execution time. We observed that in a many-core processor with fine-grained parallel workloads, the dominant source of thermal imbalance is often between the cores and the interconnection network. According to preliminary results, a DTM technique that exploits this imbalance by individually managing the interconnect can perform up to 46% better than the global DTM for irregular parallel benchmarks. This paper provides several other high-level insights on the effect of individually managing the interconnect and the computing clusters. Our analysis is a step forward from the previous work on multi-core DTM which exclusively focused on systems with a small number of relatively complex serial processors and uneven distribution of the load among these cores. Future work will extend the results presented here to a multi-tasked environment, where the XMT chip will be able to simultaneously execute multiple fine-grained parallel tasks. This extension poses an interesting optimization problem where the management algorithm will also determine the number of threads to be run from each task and how they map on the TCUs.

Acknowledgment

Partial support by NSF grants 0325393, 0811504, 0834373 and 0926237 is gratefully acknowledged.

References

1. Balkan, A.O., Horak, M.N., Qu, G., Vishkin, U.: Layout-accurate design and implementation of a high-throughput interconnection network for single-chip parallel processing. In: Proc. Hot Interconnects (2007)

2. Bell, N., Garland, M.: Implementing sparse matrix-vector multiplication on throughput-oriented processors. In: Proc. SC (2009)
3. Caragea, G., Keceli, F., Tzannes, A., Vishkin, U.: General-purpose vs. GPU: Comparison of many-cores on irregular workloads. In: Proc. HotPar (2010)
4. Che, S., Boyer, M., Meng, J., Tarjan, D., Sheaffer, J.W., Lee, S.H., Skadron, K.: Rodinia: A benchmark suite for heterogeneous computing. In: Proc. IISWC (2009)
5. Donald, J., Martonosi, M.: Techniques for multicore thermal management: Classification and new exploration. In: Proc. ISCA (2006)
6. Ge, Y., Malani, P., Qiu, Q.: Distributed task migration for thermal management in many-core systems. In: Proc. DAC (2010)
7. Ginosar, R.: The plural architecture. www.plurality.com (2011), also see course on Parallel Computing, Electrical Engineering, Technion http://webee.technion.ac.il/courses/048874
8. Hoberock, J., Bell, N.: Thrust: A parallel template library (2009), http://www.meganewtons.com/, version 1.1
9. Howard, J., Dighe, S., et al.: A 48-core IA-32 message-passing processor with DVFS in 45nm CMOS. In: Proc. ISSCC (2010)
10. Huang, W., Stan, M.R., Sankaranarayanan, K., Ribando, R.J., Skadron, K.: Many-core design from a thermal perspective. In: Proc. DAC (2008)
11. Isci, C., Martonosi, M.: Runtime power monitoring in high-end processors: Methodology and empirical data. In: Proc. MICRO (2003)
12. Kadin, M., Reda, S., Uht, A.: Central vs. distributed dynamic thermal management for multi-core processors: which one is better? In: Proceedings of the Great Lakes symposium on VLSI (2009)
13. Kaxiras, S., Martonosi, M.: Computer Architecture Techniques for Power Efficiency. Morgan and Claypool Publishers (2008)
14. Keceli, F.: Power and Performance Studies of the Explicit Multi-Threading (XMT) Architecture. Ph.D. thesis, University of Maryland (2011)
15. Keceli, F., Moreshet, T., Vishkin, U.: Power-performance comparison of single-task driven many-cores, submitted for publication.
16. Keceli, F., Tzannes, A., Caragea, G., Vishkin, U., Barua, R.: Toolchain for programming, simulating and studying the XMT many-core architecture. In: Proc. HIPS (2011), in conj. with IPDPS
17. Keceli, F., Vishkin, U.: XMTSim: Cycle-accurate Simulator of the XMT Many-Core Architecture. Tech. Rep. UMIACS-TR-2011-02, Univ. of Maryland (2011)
18. Keller, J., Kessler, C., Traeff, J.L.: Practical PRAM Programming. John Wiley & Sons, Inc., New York, NY, USA (2001)
19. Kumar, R., Hinton, G.: A family of 45nm IA processors. In: Proc. ISSCC (2009)
20. Li, S., Ahn, J.H., Strong, R.D., Brockman, J.B., Tullsen, D.M., Jouppi, N.P.: McPAT: an integrated power, area, and timing modeling framework for multicore and manycore architectures. In: Proc. MICRO (2009)
21. Liu, S., Zhang, J., Wu, Q., Qiu, Q.: Thermal-aware job allocation and scheduling for three dimensional chip multiprocessor. In: Proceedings of the International Symposium on Quality Electronic Design (2010)
22. Ma, K., Li, X., Chen, M., Wang, X.: Scalable power control for many-core architectures running multi-threaded applications. In: Proc. ISCA (2011)
23. NVIDIA: CUDA SDK 2.3 (2009), www.nvidia.com/cuda
24. Padua, D., Vishkin, U.: Joint UIUC/UMD parallel algorithms/ programming course. In: Proc. EduPar (2011), in conj. with IPDPS
25. Patterson, D.: The trouble with multicore: Chipmakers are busy designing microprocessors that most programmers can't handle. IEEE Spectrum (July 2010)
26. Satish, N., Harris, M., Garland, M.: Designing efficient sorting algorithms for manycore GPUs. In: Proc. IPDPS (2009)
27. Skadron, K., Stan, M.R., Huang, W., Velusamy, S., Sankaranarayanan, K., Tarjan, D.: Temperature-aware microarchitecture. In: Proc. ISCA (2003)
28. Wen, X., Vishkin, U.: FPGA-based prototype of a PRAM on-chip processor. In: Proc. Comp. Front. (2008)
29. Wilton, S., Jouppi, N.: CACTI: an enhanced cache access and cycle time model. IEEE J. Solid-State Circuits 31(5), 677–688 (May 1996)

Mainstream Parallel Array Programming on Cell

Paul Keir[1], Paul W. Cockshott[1] and Andrew Richards[2]

[1] School of Computing Science, University of Glasgow, UK,
[2] Codeplay Software Ltd., Edinburgh, UK,

Abstract. We present the E♯ compiler and runtime library for the 'F' subset of the Fortran 95 programming language. 'F' provides first-class support for arrays, allowing E♯ to implicitly evaluate array expressions in parallel using the SPU co-processors of the Cell Broadband Engine. We present performance results from four benchmarks that all demonstrate absolute speedups over equivalent 'C' or Fortran versions running on the PPU host processor. A significant benefit of this straightforward approach is that a serial implementation of any code is always available, providing code longevity, and a familiar development paradigm.

1 Introduction

Collection-oriented programming languages [1] are characterised by the provision of a built-in selection of operations to manipulate aggregate data structures in a holistic manner. Idiomatic code in these languages will commonly eshew the use of loop constructs. The potential to extract parallelism from this style of programming is consequently, and firstly, due to the *divisibility* of these aggregate data types; and secondly to the lack of side-effects in the expressions or constructs which stand in place of the imperative loops. Collection-oriented programming has often been applied to distributed parallel architectures, however it is just as relevant in the setting of heterogeneous multicore.

A perennial concern of performance-critical code structured around imperative loops appears within the context of implicit, or automatic, parallelism. An auto-parallelising compiler faced with a side-effecting loop which exhibits a sequential execution semantics, may overcome the challenge by a code transformation which introduces parallelism, along with locks or semaphores. The user of such a compiler is soon compelled to understand a new layer of diagnostic messages, which gradually cajole them towards an alternative, highly structured, coding style. The resulting code will often be data-parallel, and specify behaviour equivalent to that common to collection languages, though considerably more verbose.

In this paper we present a mainstream solution for scientific computing in the auto-parallelising array compiler, E♯, which targets the heterogeneous architecture of the Cell Broadband Engine. We also discuss the design decisions behind our implementation of four classic benchmarks[1] , before presenting an analysis of performance experiments.

1.1 Related Work

The seminal array language, originating in the 1960s, is Kenneth Iverson's APL. Subsequent decades brought a number of *parallel* array languages, for *distributed architec-*

[1] Available at `http://www.dcs.gla.ac.uk/people/personal/pkeir/hppc11code.7z`

tures: HPF, NESL, and ZPL being notable examples. Recent trends towards multicore systems have brought about a renaissance in the design of parallel array languages.

Single Assignment C (SAC) is a pioneering functional array research language based on the syntax and semantics of 'C'. SAC is distinguished by its first-class arrays, absent pointers and side-effects, and an advanced typing system capable of shape-polymorphic array function definitions. SAC has also recently targeted the heterogeneous GPU architecture via a CUDA backend [2]. The absence of a stack in CUDA however necessitates that no function calls are present within the SAC WITH-loops which provide the sites for parallelisation.

Cray's Chapel language, and IBM's X10, were the two finalists from the ten year DARPA High Productivity Computing Systems (HPCS) [3] programme. Both use a partitioned global address space (PGAS) model, and allow high-level, holistic, and parallelisable manipulation of arrays. The two standard implementations of these languages target distributed parallel architectures.

Microsoft's Accelerator project [4] targets homogeneous x86, and heterogeneous GPU architectures, using a data-parallel .NET array library which, by delaying the evaluation of array expressions, can minimise the creation of intermediate structures.

By virtue of the highly expressive Haskell typing system, the Repa [5] parallel array *library*, is refreshingly akin to an embedded array *language*. Absolute parallel performance comparable to serial 'C' derives from optimisations such as array fusion; and mandatory unboxed, strictly evaluated array elements. For the end user, performance can still depend on careful application of the force function; which replaces a *delayed* with a *manifest* array. Repa builds on the long-running Data-Parallel Haskell research strand, and for now targets only homogeneous multicore systems.

2 Implicit Parallelism using the 'F' Programming Language

Fortran was originally developed by John Backus and others at IBM in the 1950s. Like 'C', Fortran is a statically typed, imperative language. Fortran has historically differentiated itself from 'C' by its absent pointer arithmetic; longstanding support for complex numbers; argument passing by reference; and with *Fortran 90*, first class array types. The Fortran language is ISO standardised, and *Fortran 2008* has been approved. Of the mainstream programming languages, Fortran has distinguished itself within the field of computational science, due to its relatively high level, and excellent performance profile.

The 'F' programming language is a subset of *Fortran 95* designed with the intention of providing a lightweight version of Fortran, free of the requirement to support 40 years of language artifacts. The primary motivation of the language design was to create a Fortran-based language for education, however 'F' is a perfectly adequate general-purpose language. Furthermore, any Fortran compiler will compile a program conforming to the 'F' language standard, the g95 compiler also has a command line switch to enable error messages.

Having the requisite support for arrays, the 'F' programming language is therefore a suitable language to explore the use of array expressions as a mechanism to drive implicit parallelism for scientific computing on a heterogeneous architecture such as the Cell Broadband Engine.

2.1 A Language Primer

The following code excerpt demonstrates an entire 'F' program, equivalent to a 'C' *main* function. The assignment on line 3 will *pointwise* multiply the elements of the two arrays bound by `b` and `c`, before adding the result to a third array induced by the literal `3`. Once all operations on the right hand side of the assignment are completed, the result is copied to the array bound to `a`.

```
1 program p
2   real, dimension(2,3) :: a, b = 1, c = 2
3   a = b * c + 3
4   a = muladd(b,c,3)
5 end program p
```

Fig. 1: Assignments involving intrinsic and user defined elemental functions.

The dimension attribute specification on line 2 of Figure 1 declares three arrays, each with an explicit *shape* vector of 2;3. The length of an array's shape vector provides another useful metric: its *rank*, and is therefore 2 in this case. The terms in an array expression must all have equal shape, and in doing so, the shapes are said to *conform*. Scalar values, such as the numeric literals `1`, `2` and `3`, are promoted, or lifted, to an array type of conforming shape, when their context within an expression requires it. The induced array is then populated by elements of the same value as the inducing scalar. In Figure 1, the expression `b * c + 3` is therefore an array expression with a rank of 2, and shape of 2;3. This is in fact true of all the expressions in Figure 1.

For the E♯ compiler, an array expression involving one or more functions, or operators, will be evaluated in parallel. The array expression `b * c + 3` from Figure 1 will therefore qualify, and so trigger the appropriate compiler transformations to ensure a parallel execution.

Scalar functions free of side-effects may also be applied to array arguments with conforming shapes. Such functions in Fortran are classified as *elemental*. The call to `muladd` on line 4 of Figure 1 represents a user defined `elemental` function producing the same result as the elemental arithmetic expression on line 3.

Unlike many auto-parallelising compilers, the E♯ user has the certainty that *all* array expressions will execute in parallel. Consequently, other iterative constructs of the 'F' language, such as `do` or `while` loops remain useful. Such constructs should be used where there is insufficient work to justify the small cost of thread adminstration and direct memory access (DMA) operations.

3 The E♯ compiler

E♯ is a source to source compiler, translating from the 'F' subset of Fortran 95 to Offload C++ [6]: a C++ language extension utilising pointer locality. The compiler targets heterogeneous multicore architectures, and in particular the CBE. The 'F' language has a

```
1 offloadThread_t tid = offload {
2   int outer *po = &g;
3   int i = *po;
4   int inner *pi = &i;
5   *pi = *pi+1;
6   *po = i;
7 };
8
9 offloadThreadJoin(tid);
```

Fig. 2: A simple asynchronous offload block expression

large standard library, and this is made available to both the PPU and the SPU using the GNU Fortran runtime libraries. A C++ template class has also been developed which both abstracts over the multifarious internal array representations of essentially all Fortran compilers; and is also compatible with the dual memory address hierarchy exposed by Offload C++. The E♯ compiler is written in the pure functional programming language Haskell. Haskell's Parsec parsing library allowed the structure of the published 'F' grammar to be followed exactly, while the Scrap Your Boilerplate package was used to perform the crucial transformations of the abstract syntax trees.

3.1 Targeting Offload C++

E♯ translates from 'F' to Offload C++, a C++ language extension and runtime library [6] targeting heterogeneous architectures. The most prominent language feature of Offload C++ is the *offload block* which provides a traditional 'C' compound statement, prefixed with the keyword `offload`, to be executed asynchronously to the main thread. Running on the CBE, each new thread will be executed by the next available SPU. An offload block returns an integer thread identifier and, like Pthreads, performance parallelism is achieved through the launch of multiple threads; subsequently joined with a call to `offloadThreadJoin`. A related benefit of this approach, is *automatic call-graph duplication*: with little or no annotation, a function, or variable reference, defined once, may be used both outside and inside an offload block.

Equally significant is the extension of the C++ type system to allow statically assigned pointer locality. In Offload C++ a pointer is, either implicitly or explicitly, identified either with an *inner* or *outer* locality. Pointer arithmetic and assignment between those of differing localities is statically prohibited by the compiler. More proactively, the dereferencing of an *outer* pointer from within an offload block corresponds to a DMA transfer from main memory to SPU scratch memory; while assignment to an *outer* pointer results in a DMA transfer in the opposite direction. Figure 2 demonstrates the concept: assuming the variable `g` is defined at global scope, the resulting effect of the offload block is for `g` to be incremented by one. Note that the `inner` and `outer` pointer qualifiers on lines 2 and 4 in Figure 2 are optional and would be automatically inferred.

```
1 template <typename T, int Od>
2 struct PtrWrapper {
3   T inout(Od) *m_p;
4 };
```

Fig. 3: Offload C++ template struct with pointer member

Flexibility in the locality of class or struct pointer members may be obtained using the static `inout` qualifier and an integer template parameter, as shown in Figure 3.

3.2 A C++ Template Interface to Fortran Runtime Libraries

Neither the 'F' nor Fortran language standards specify an application binary interface (ABI) for arrays. With over a dozen Fortran compilers it would be unfortunate to restrict the E♯ compiler to only one of the associated runtime libraries. The Chasm project [7] helps addresses this issue by providing a low-level 'C' API targeting the internal "dope vectors" used by each Fortran compiler. For E♯, two new C++ array template classes have been designed and implemented: `ArrayT`; and the statically sized `ArrayTN`. Each provides high-level support for Fortran array features such as sectioning; serialisation; de-serialisation; and fast indexing with optional non-`1` lower bound.

3.3 Parallel Operational Semantics

The E♯ compiler attempts a human-readable, one-to-one correspondence between 'F' input and C++ output language constructs. An exception occurs at a parallelised array expression. In this instance, the array expression is transformed into a nested `for` loop[1], with depth equal to an expression's rank. A team of threads is then launched, each assigned a statically allocated and contiguous chunk of the outermost iteration space. The precise number of threads is set on program startup using an environment variable, `ESHARP_NUM_THREADS`, and may range from 1 to 128. Each individual thread is given the full resources of an SPU, and sits in a notional FIFO queue until one is available. While it can be assumed that launching one thread for each SPU will incur the lowest thread administration costs, while maximising resource usage, a program with a large working set may need to be split into more than six pieces. For example, an array expression with a 6000KiB working set, will exceed the 256KiB local store of an SPU if partitioned across six threads. With 32 or more threads, the program should run.

4 The Benchmark Programs

The first two benchmark programs we will examine, BlackScholes and Swaptions, are financial simulations from Princeton Univerity's PARSEC benchmark suite, converted by hand to 'F' from original C and C++. Our Mandelbrot program allows us to look at DMA transfer bottlenecks. while exploring differing approaches to parallel decomposition. Finally, a simulation of the n-body problem is examined.

[1]The operation is also recursively applied to array subexpressions.

4.1 Blackscholes

Blackscholes is a financial simulation which prices a portfolio of options using a partial differential equation now known as the *Black-Scholes equation*. Scalability in performance is obtained using a chunked, fine-grained decomposition, and calculating multiple options in parallel. The original implementation of Blackscholes uses Threading Building Blocks (TBB) and Pthreads to facilitate parallelism, with both using an *array of structs* configuration. Beneath the requisite file IO and threading boilerplate, there are two functions within the call graph of the parallel region: `BlkSchlsEqEuroNoDiv`, and a "callee" function `CNDF`. The kernel is given 100 runs, each of which is launched by an application of the TBB `parallel_for` template. This invokes multiple calls to a user-defined worker class's overloaded function operator. The 'F' version requires only that we mark the function as `elemental`, and the kernel launch is then
`prices = BlkSchlsEqEuroNoDiv(dat)`

4.2 Swaptions

The Swaptions program prices a portfolio of interest-rate swap options by the Heath-Jarrow-Morton framework using Monte-Carlo simulation. The original program consists of around fifteen C++ source files, then converted to 'F'.

Parallel decomposition on both TBB and Pthread implementations was, like Blackscholes, static and course-grained, though distinguished by a significantly larger working set. An *array of structs* configuration was again present in the C++ code, and the kernel was again dominated by a single 16-parameter function, `HJM_Swaption_Blocking`, applied in parallel to chunks from an one-dimensional iteration space.

The `HJM_Swaption_Blocking` function was ultimately a suitable target for `elemental` status, however the element type of two of its arguments are pointers to 1D and 2D arrays. An 'F' `elemental` function cannot accept arrays as a "scalar" element type, so necessitating the definition of two array wrapper types. With the 1D *pdYield* array, this amounts to the type shown in Figure 4.

```
1 type, public :: yieldT
2   real(kind=ki), dimension(m_iN) :: y
3 end type yieldT
```

Fig. 4: A scalar 'F' datatype wrapping an array

The C++0x code generated by E♯ from Figure 4 is shown in Figure 5. Notice that the `struct` has a template parameter, used to specify the *locality* of the data accessed via the `ArrayTN` member at line 18. That this is an `ArrayTN`, rather than an `ArrayT`, is an automatic optimisation due to the `m_iN` from line 2 of Figure 4 being a compile-time constant; the integer template argument `11` specifies the statically-allocated data size.

```
1  template <int Od>
2  struct yieldT {
3    inline yieldT () {};
4    inline yieldT (const ArrayTN<__compiler,float,1,11,Od> &&y)
5      : y(y) {};
6    inline friend ostream & operator << (ostream &o,
7                                          const yieldT<Od> &t) {
8      o << t.y; return o;
9    };
10   inline friend istream & operator >> (istream &i,
11                                         yieldT<Od> &t) {
12     i >> t.y; return i;
13   };
14   template <int Od2>
15   inline yieldT &operator= (const yieldT<Od2> &rhs) {
16     y = rhs.y; return *this;
17   };
18   ArrayTN<__compiler,float,1,11,Od> y;
19 };
```

Fig. 5: The C++ struct generated from Figure 4 by E♯

4.3 Mandelbrot

Estimation of the Mandelbrot set requires iteration of the complex function $z_{n+1} = z_n^2 + c$. Of the two Mandelbrot benchmarks we have developed, the first is more straightforward. An array of the same size as the 8-bit output image is initialised with positive integer coordinate pairs within the appropriate range, leaving the `elemental` function to create the complex value upon which it iterates. A second, blocked, version of the program partitions the coordinate array into squares. A user defined type is used for the squares, and is the scalar type upon which the requisite `elemental` function is defined.

4.4 The n-Body problem

From earlier work [8] we were aware that an $O(n^2)$ "all-pairs" n-body simulation on CBE can exhibit good scaling at the expense of wall clock time, and so a tiled decomposition of the problem, inspired by research at Nvidia [9], was developed.

The kernel of our n-body algorithm performs the $O(n^2)$ force calculation in parallel while the remaining leapfrog-Verlet integration updates the positions and velocities, and is run in serial by the host processor. This choice seems reasonable as having only linear complexity, the percentage of runtime expended on the remaining integration stage becomes insignificant with larger body counts. A square shaped tile of the pairwise body interactions, maximises the number of calculations that can be performed per body. That is to say, a DMA transfer of $2p$ body positions and masses, will provide p^2 components of force for the integrator.

The E♯ compiler parallelises only the outermost of the generated loops. To fully exploit the two-dimensional decomposition already outlined, a "flattened", *one-dimensional*,

array is used to feed the requisite driving `elemental` function. User-defined scalar types, are once again required for the input and output elements. For input and output respectively the two types `pchunk2d` and `accel_chunk` are shown in Figure 6.

```
1 type, public :: pchunk2d
2   type(vec4), pointer, dimension(:) :: ivec4, jvec4
3 end type pchunk2d
4
5 type, public :: accel_chunk
6   type(vec3), dimension(CHUNK_SIZE) :: avec3
7 end type accel_chunk
```

Fig. 6: The n-body kernel input (`pchunk2d`) and output (`accel_chunk`) wrapper types

5 Experimental Evaluation

The following benchmark results were measured and averaged across five runs on a PlayStation 3 running Fedora Core 7. Single-precision was used throughout, due to the CBE's slow double-precision execution. In addition to the 4.1.1 versions of the GNU C, C++, and Fortran compilers provided with the installed IBM Cell SDK v3.0, version 4.6 of GCC is also installed. Where a speedup metric is presented, the fastest available PPU serial version is used, with selection based on source language; compiler; and the often powerful GCC switch: `-mcpu=cell`. The Offload C++ compiler version is 2.0.2, patched to use SPU GCC 4.6. All compilers use the `-O3` switch throughout.

BlackScholes This benchmark exceeds the memory limitations of the SPU at low thread counts. However, with 18 threads the E♯ version outperforms GCC after 4K options. With 64 threads, 256K options become possible, and provide a final speedup of over 11; shown in Figure 7. The surprisingly horizontal E♯ curves indicate that the problem is dominated by thread administration. The serial results also demonstrate that the 'F' version performs competitively with the independently constructed 'C' version.

Swaptions Though speedup values also increased, slightly, with input data size, Figure 8 shows only a maximum speedup of around 2.3x over GCC 4.6 with the "Large" data set. Lower thread counts for Swaptions are possible, and the graph demonstrates good scaling to 6 threads. While greater thread counts, up to 128, are shown as redundant in this configuration, it is encouraging to see that increasing thread administration overheads have no noticeable effect, as no fall in speedup is observed.

Mandelbrot As anticipated, the blocked version of Mandelbrot, using 64x64 squares, outperformed the naïve version, presumably due to reduced DMA traffic. The blocked version was also able to create a 2048x2048 image, and so achieve a speedup of almost 12x. Mandelbrot reaches a similar maximum speedup as Blackscholes; though a distinctive fall-off is also observed. See Figure 9.

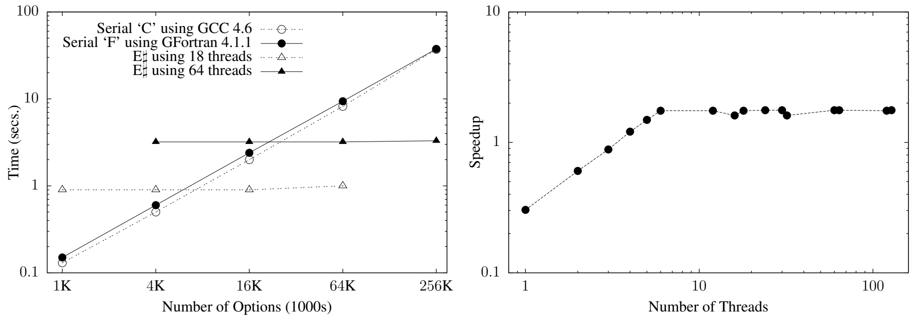


Fig. 7: Wallclock Blackscholes Kernel Timings for 100 iterations

Fig. 8: E♯ Speedup with Swaptions and "Large" data set

The n-Body Problem Using 16x16 square tiles speedups increase gradually with data sizes, reaching a 3.4x speedup against the fastest 'C' configuration on PPU; which uses the older GCC 4.1.1 and the `-mcpu=cell` switch. In comparison to times obtained from the PPU only, using GFortran 4.1.1 `-mcpu=cell`, and the same 'F' code, a speedup of 4.9x is obtained with 16384 bodies; shown in Figure 10.

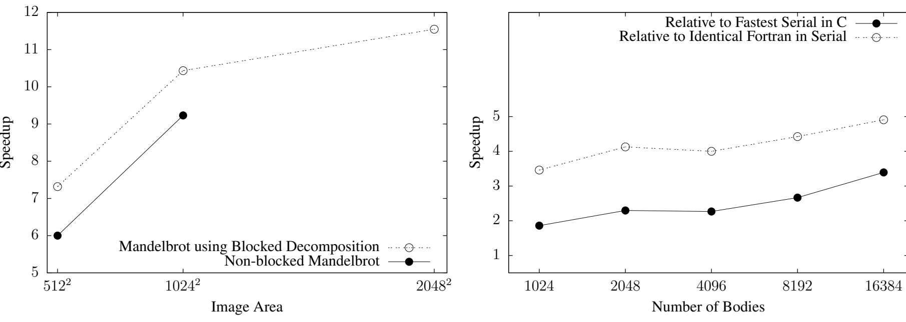


Fig. 9: Relative Mandelbrot Speedups against Image Area

Fig. 10: Relative n-Body Speedups against Input Data Size

6 Conclusion

We have demonstrated the auto-parallelising array compiler, E♯, targeting the heterogeneous architecture of the Cell Broadband Engine. Encouraging performance results from four benchmarks are presented, and show speedups ranging from 2-11x over serial versions running on PPU only. The language employed, 'F', is a simple, useable, and

standard dialect of modern Fortran, and is therefore well positioned for expected users from the scientific programming community. In addition, 'F' codes developed for use by E♯ are also valid Fortran; and shown to perform competitively in serial.

E♯ would benefit from the inclusion of streaming, rather than the current static partitioning of the iteration-space. This should allow access to a larger range of problem sizes, and hopefully more routine access to high performance. Also, as array expressions are free of side-effects, we can expose a finer level of granularity than currently offered by E♯, which presently partitions only the outermost rank. This should help load balancing on problems with small outer rank extents.

The techniques described here for the CBE could also be applied to new multicore processors such as Intel's Single-Chip Cloud Computer (SCC), or Knight's Corner. In the case of the SCC it would be possible to produce an E♯ compiler provided that a version of the Offload system were ported to the SCC. This is likely to result in somewhat lower performance than the Cell because of 3 factors: a) processors on the SCC can not initiate reads from host memory, b) inter-process communication on the SCC relies on a software library, RCCE, rather than the CBE's DMA hardware; c) the performance of the inidividual SCC processors is slower than the host Xeon, whereas the Cell SPUs are capable of higher throughput than the host PPC. For shared memory machines like Knight's Corner, we anticipate implementing E♯ by compiling to C++ with OpenMP pragmas. Some prototype work has already been done using this approach.

References

1. J. Sipelstein and G. E. Blelloch, "Collection-Oriented Languages," *Proceedings of the IEEE*, vol. 79, pp. 504–523, 1991.
2. J. Guo, J. Thiyagalingam, and S.-B. Scholz, "Breaking the GPU programming barrier with the auto-parallelising SAC compiler," in *Proceedings of the sixth workshop on Declarative aspects of multicore programming*. ACM Press, 2011, pp. 15–24.
3. M. Weiland, "Chapel, Fortress and X10: novel languages for HPC," EPCC, The University of Edinburgh, Tech. Rep. HPCxTR0706, October 2007.
4. D. Tarditi, S. Puri, and J. Oglesby, "Accelerator: Using Data Parallelism to Program GPUs for General-Purpose uses," in *Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*. ACM Press, 2006, pp. 325–335.
5. G. Keller, M. M. Chakravarty, R. Leshchinskiy, S. Peyton Jones, and B. Lippmeier, "Regular, shape-polymorphic, parallel arrays in Haskell," in *Proceedings of the 15th ACM SIGPLAN international conference on Functional programming*. ACM Press, 2010, pp. 261–272.
6. P. Cooper, U. Dolinsky, A. F. Donaldson, A. Richards, C. Riley, and G. Russell, "Offload - Automating Code Migration to Heterogeneous Multicore Systems," in *Proceedings of the 5th International Conference on High Performance and Embedded Architectures and Compilers*, vol. 5952. Springer, 2010, pp. 337–352.
7. C. E. Rasmussen, M. J. Sottile, S. S. Shende, and A. D. Malony, "Bridging the language gap in scientific computing: the Chasm approach," *Concurrency and Computation: Practice and Experience*, vol. 18, pp. 151–162, 2006.
8. A. F. Donaldson, P. Keir, and A. Lokhmotov, "Compile-time and Run-time Issues in an Auto-parallelisation System for the Cell BE Processor," in *Proceedings of the 2nd EuroPar Workshop on Highly Parallel Processing on a Chip*, vol. 5415. Springer, 2008, pp. 163–173.
9. M. H. Lars Nyland and J. Prins, "Fast N-Body Simulation with CUDA," in *GPU Gems 3*, H. Nguyen, Ed. Addison-Wesley Professional, 2007, pp. 677–694.

Generating GPU Code from a High-level Representation for Image Processing Kernels

Richard Membarth[1]*, Anton Lokhmotov[2], and Jürgen Teich[1]

[1] Hardware/Software Co-Design, Department of Computer Science,
University of Erlangen-Nuremberg, Germany.
{richard.membarth,teich}@cs.fau.de

[2] Media Processing Division, ARM,
Cambridge, United Kingdom.
anton.lokhmotov@arm.com

Abstract. We present a framework for representing image processing kernels based on decoupled access/execute metadata, which allow the programmer to specify both execution constraints and memory access pattern of a kernel. The framework performs source-to-source translation of kernels expressed in high-level framework-specific C++ classes into low-level CUDA or OpenCL code with effective device-dependent optimizations such as global memory padding for memory coalescing and optimal memory bandwidth utilization. We evaluate the framework on several image filters, comparing generated code against highly-optimized CPU and GPU versions in the popular OpenCV library.

1 Introduction

Computer systems are increasingly heterogeneous, as many important computational tasks, such as multimedia processing, can be *accelerated* by special-purpose processors that outperform general-purpose processors by 1–2 orders of magnitude, importantly, in terms of energy efficiency as well as in terms of execution speed.

Until recently, every accelerator vendor provided their own application programming interface (API), typically based on the C language. For example, NVIDIA's API called CUDA C [6] targets systems accelerated with Graphics Processing Units (GPUs). In CUDA, the programmer dispatches compute-intensive data-parallel functions (*kernels*) to the GPU, and manages the interaction between the CPU and the GPU via API calls. Ryoo et al. [7] highlight the complexity of CUDA programming, in particular, the need for exploring thoroughly the space of possible implementations and configuration options. OpenCL [8], a new industry-backed standard API that inherits many traits from CUDA, aims to provide software portability across heterogeneous systems: correct OpenCL programs will run on any standard-compliant implementation. OpenCL per se, however, does not address the problem of *performance portability*; that is, OpenCL code optimized for one accelerator device may perform dismally on another, since performance may significantly depend on low-level details, such as data layout and iteration space mapping [4].

* This work was partly done during the author's internship at ARM, which was sponsored by the European Network of Excellence on High Performance and Embedded Architectures and Compilation (HiPEAC).

Low-level programming increases the cost of software development and maintenance: whilst low-level languages can be robustly compiled into efficient machine code, they effectively lack support for creating portable and composable software.

High-level languages with domain-specific features are more attractive to domain experts, who do not necessarily wish to become target system experts. To compete with low-level languages for programming accelerated systems, however, domain-specific languages should have an acceptable performance penalty.

We present a framework for image processing that allows programmers to concentrate on developing algorithms and applications, rather than on mapping them to the target hardware. While previous work shows that running the same kernels (e. g., written in OpenCL) on different hardware (from AMD and NVIDIA) can have significant impact on the performance [3], this framework serves to protect investments in software in the face of the ever changing landscape of computer systems.

The framework is implemented as a library of C++ classes (§2.1) and a Clang-based compiler producing host and device code in CUDA C and OpenCL (§2.2). Our framework is most similar in spirit to Cornwall *et al.*'s work on indexed metadata for visual effects [2] but introduces additional device-specific optimizations such as global memory padding for memory coalescing and optimal bandwidth utilization. We evaluate the framework by comparing generated code against highly-optimized CPU and GPU versions in the popular OpenCV library (§3).

2 Image Processing Framework

Our framework provides a library of C++ classes for representing image processing kernels (§2.1) and a source-to-source compiler for translating library constructs into host and device code in CUDA or OpenCL (§2.2). The library is based on the concept of decoupled access/execute metadata, which capture both execution constraints and memory access patterns of a kernel [4]. The compiler is built using Clang [1], an open source frontend for C-family languages.

2.1 Library

The library consists of built-in C++ classes that describe the following three basic components required to express image processing on an abstract level:

- *Image*: Describes data storage for the image pixels. Each pixel can be stored as an integer number, a floating point number, or in another format such as RGB, depending on instantiation of this templated class. The data layout is handled internally using multi-dimensional arrays.
- *Iteration Space*: Describes a rectangular region of interest in the output image, for example the complete image. Each pixel in this region is a point in the iteration space.
- *Kernel*: Describes an algorithm to be applied to each pixel in the region of interest.

These components are an instance of decoupled access/execute metadata [4]: the *Iteration Space* specification provides ordering and partitioning constraints (execute metadata); the *Kernel* specification provides a pattern of accesses to uniform memory (access metadata). Currently, the access/execute metadata is mostly implicit: we assume that the iteration space is parallel in all dimensions and has a 1:1 mapping to work-items (threads), and that the memory access pattern is obvious from the kernel code.

Example We illustrate our image processing framework using a grayscale vertical mean image filter, for which the output pixel with coordinates (x,y) is the average of D input column pixels:

$$\mathbf{O}_{x,y} = \frac{1}{D}\sum_{k=0}^{D-1}\mathbf{I}_{x,y+k}, \text{ where } 0 \le x < W, 0 \le y < H-D. \tag{1}$$

- $\mathbf{I}$ is an input image of $W \times H$ pixels;
- $\mathbf{O}$ is an output image of $W \times H$ pixels;
- D is the *diameter* of the filter, that is, the number of input pixels over which the mean is computed (typically, $D \ll H$).

To express this filter, the framework user derives a class from the built-in *Kernel* class and implements the virtual *kernel* function, as shown in Listing 1. The *kernel* function (line 10) takes an *ElementIterator* argument that represents the output pixel for which the algorithm is run. To access the pixels of an image, the parenthesis operator `()` is used, taking the *ElementIterator* argument as a mandatory parameter, and the column (`dx`) and row (`dy`) offsets as optional parameters. The user instantiates the class with input and output images, an iteration space, and other parameters that are member variables of the class.

```
1  class VerticalMeanFilter : public Kernel {
2    private:
3      Image<float> &Input, &Output;
4      int d;
5
6    public:
7      VerticalMeanFilter(IterationSpace &IS, Image<float> &Input,
           Image<float> &Output, int d) :
8        Kernel(IS), Input(Input), Output(Output), d(d) {}
9
10     void kernel(IterationSpace::ElementIterator EI) {
11       float sum = 0.0f;
12
13       for (int k=0; k<d; ++k) {
14         sum += Input(EI, 0, k);
15       }
16
17       Output(EI) = sum/(float)d;
18     }
19 };
```

Listing 1: The vertical mean filter expressed in our framework.

In Listing 2, the input and output *Image* objects `IN` and `OUT` are defined as two-dimensional $W \times H$ grayscale images, having pixels represented as floating-point numbers (lines 8–9). The *Image* object `IN` is initialized with the `host_in` pointer to a plain C array with raw image data, which invokes the = operator of the *Image* class (line 12). The region of interest `VIS` contains all image columns but excludes the last *d* rows to simplify border handling in this example (line 15). The kernel is initialized with the

iteration space object, image objects and kernel diameter d (line 18), and executed by a call to the *execute()* method (line 21). To retrieve the output image, the `host_out` pointer to a plain C data array is assigned the *Image* object `OUT`, which invokes the `getData()` operator (line 24).

```
1  const int width = 5120, height = 3200, d = 40;
2
3  // pointers to raw image data
4  float *host_in = ...;
5  float *host_out = ...;
6
7  // input and output images
8  Image<float> IN(width, height);
9  Image<float> OUT(width, height);
10
11 // initialize input image
12 IN = host_in; // operator=
13
14 // define region of interest
15 IterationSpace VIS(width, height-d);
16
17 // define kernel
18 VerticalMeanFilter VMF(VIS, IN, OUT, d);
19
20 // execute kernel
21 VMF.execute();
22
23 // retrieve output image
24 host_out = OUT.getData();
```

Listing 2: Example code that initializes and executes vertical mean filtering.

2.2 Compiler

This section describes the design of our source-to-source compiler and the single steps taken to create CUDA C and OpenCL code from a high-level description of image objects, iteration space objects and kernel objects.

Our source-to-source compiler is based on the latest Clang/LLVM compiler framework. The Clang frontend for C/C++ is used to parse the input files and to generate an AST representation of the source code. Our backend uses this AST representation to generate host and device code in CUDA or OpenCL.

Kernel Code The compiler creates the kernel code AST in multiple steps.

First, the kernel declaration is created. The kernel parameters are identified from the *Kernel* class constructor. Each variable, reference, or pointer has to be initialized in the constructor of the *Kernel* class and a corresponding kernel parameter is added to the declaration. In doing so, references to image objects are replaced by global memory pointers to the pixel type. The existing kernel method argument—the *ElementIterator*—is removed. Some additional parameters such as the image width and height are added for index calculations and for future uses like border handling.

Second, the kernel body is created from the *kernel* method of the class. To get an AST for the kernel body, the original AST is copied with certain AST nodes replaced. References to *Image* objects are replaced with references to corresponding arrays. Instead of using the *ElementIterator* to calculate the image index, the compiler adds statements at the beginning of the kernel that calculate the pixel location from the thread index and block index in CUDA C or the global indices in OpenCL. Similarly, each class member expression – access to a member variable of the kernel class – is translated to a reference to the corresponding kernel function parameter. After the translation, we get an AST that can be used for further transformations.

After transformations, the AST is pretty printed and stored to a file. During pretty printing, CUDA C and OpenCL C specific function and variable qualifiers are emitted. For example, the `__global__` qualifier in CUDA C and the `__kernel` qualifier in OpenCL are emitted for entry functions.

Host Code Unlike for device code, we create no AST for host code. Rather, we use Clang's *Rewriter* functionality to change the textual representation of AST nodes, whilst leaving the nodes intact.

To invoke previously generated device kernels, the framework code gets translated into corresponding CUDA or OpenCL API calls as follows:

– *Image declarations* (line 8 and 9): Get translated into device memory allocation using `cudaMalloc` or `clCreateBuffer`.
– *Memory assignments* (line 12): Get translated into memory transfers using `cuda-Memcpy` or `clEnqueueWriteBuffer`.
– *IterationSpace declaration* (line 15): Defines the kernel execution configuration.
– *Kernel declaration* (line 18): Gets translated into loading the kernel source. For CUDA C, this step is not required. For OpenCL, the kernel source is loaded from a file, an OpenCL program for the loaded source is created, and the kernel is compiled.
– *Kernel execution* (line 21): Gets translated into launching the kernel, using the execution configuration obtained from from the corresponding `IterationSpace` declaration.
– *Memory assignments* (line 24): Get translated into memory transfers using `cuda-Memcpy` or `clEnqueueReadBuffer`.

In addition to the above changes, further changes are required to get proper CUDA C or OpenCL code. First of all, include directives for the CUDA C or OpenCL headers are added. In addition, the CUDA C kernel sources are included at the beginning of the file. To initialize the runtime, we add the corresponding functionality at the beginning of the main function. In particular, for OpenCL this initialization is an important part since it sets up the platform and device to be used for execution. After these changes, the generated host and device files can be compiled.

Padding Support To avoid conflicts in accessing image pixels in global memory, our compiler adds padding to allocated host and device memory. For host code, special memory allocation functions are required to allocate memory so that each image row is padded to get the desired data alignment. For device code, array index calculations are changed to take padding into account. The compiler handles padding automatically, given the desired alignment amount for the target device.

3 Results

3.1 Vertical Mean Filtering

A naïve parallel algorithm can run $N = W \times (H - D)$ threads, each producing a single output element, which requires $\Theta(ND)$ reads and arithmetic operations. A good parallel algorithm, however, must be efficient and scalable [5]. Therefore, we use an algorithm that *strips* the computation, where up to T outputs in the same strip are computed serially in two *phases* [4]: The first phase computes $\mathbf{O}_{x,y0}$ according to (1), while the second phase computes $\mathbf{O}_{x,y}$ for $y \geq y0+1$ as $\mathbf{O}_{x,y-1} + \left(\mathbf{I}_{x,y+D-1} - \mathbf{I}_{x,y-1}\right)/D$.

This algorithm performs $\Theta(N + ND/T)$ reads and arithmetic operations, considerably reducing memory bandwidth and compute requirements for $T \gg D$, whilst allowing up to $\lceil N/T \rceil$ threads to run in parallel. Thus, this algorithm trades off work efficiency against parallelism.

Listing 3 shows the implementation of this algorithm in our framework. Since our framework supports currently only a 1:1 mapping of output pixels to threads, we use the offset specification to calculate the pixel location for a 1:N mapping. We will provide special syntax for a 1:N mapping in the future.

```
1  class VerticalMeanFilterRollingSum : public Kernel {
2    ...
3    void kernel(IterationSpace::ElementIterator EI) {
4      float sum = 0.0f;
5      int t0 = EI.getY();
6
7      // first phase: convolution
8      for (int k=0; k<d; ++k) {
9        sum += Input(EI, 0, k + (t0*NT-t0));
10     }
11     Output(EI, 0, (t0*NT-t0)) = sum/(float)d;
12
13     // second phase: rolling sum
14     for (int dt=1; dt<min(NT, height-d-(t0*NT)); ++dt) {
15       int t = (t0*NT-t0) + dt;
16       sum -= Input(EI, 0, t-1);
17       sum += Input(EI, 0, t-1+d);
18       Output(EI, 0, t) = sum/(float)d;
19     }
20   }
21 };
```

Listing 3: Kernel description of the vertical mean filter using a rolling sum.

We compare the performance of code generated by our framework against that of hand-written code reported in [4].[3] We run the vertical mean filter with different values for T, that is, changing the number of pixels calculated by one thread. Figure 1 shows the execution times of the vertical mean filter applied to an image of 5120×3200 pixels. Processing more than one pixel increases the throughput from 0.53 Gpixel/s for $T = 1$, up to the peak throughput of 6.6 Gpixel/s at several points (e. g., for $T = 528$).

[3] We use the same configuration: thread block dimensions 128×1, kernel diameter $D = 40$. However, we use Quadro FX 5800, rather than GTX 280.

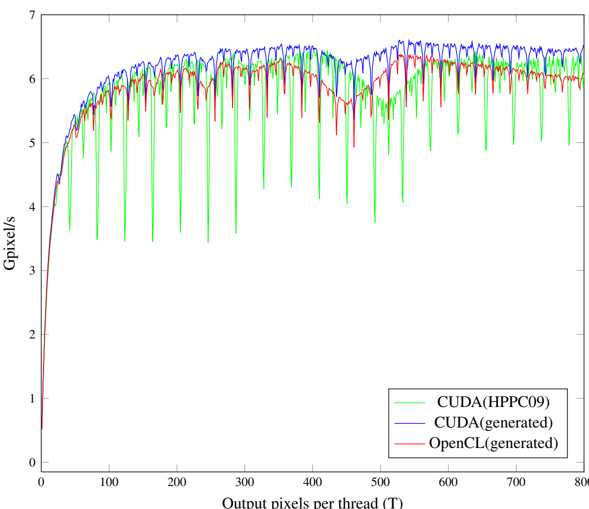


Fig. 1: Throughput of the generated CUDA C/OpenCL sources in Gpixel/s for the vertical mean filter on an image of 5120×3200 pixels in comparison to the hand-written CUDA code from [4].

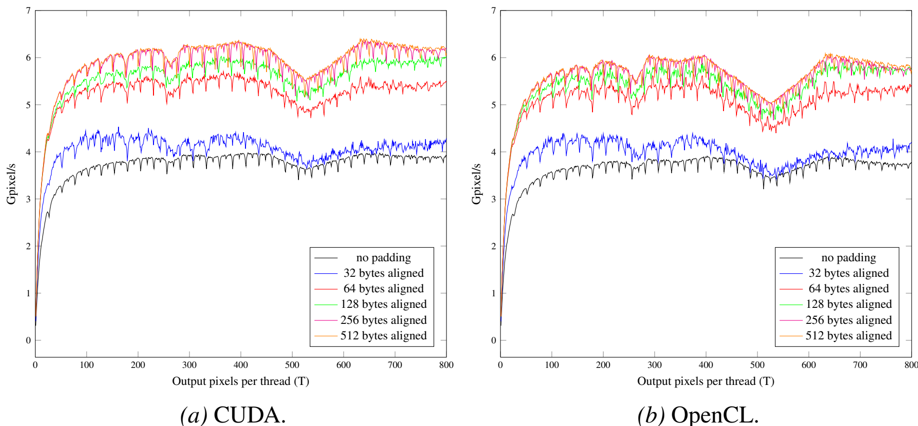
The results show that the generated CUDA code achieves the same performance as the optimized hand-written CUDA code.[4] However, our high-level implementation is concise and has only a fraction of the complexity of the low-level implementation of [4]. For instance, in terms of lines of code, the low-level implementation consists of about 500 lines of host and device code, whilst the high-level implementation consists of fewer than 50 lines of code.

In the previous example, the image width of 5120 is a multiple of the SIMD width of the underlying hardware (which is 32). This results in optimal memory transfers utilizing memory bandwidth best. However, if the image width is not a multiple of the SIMD width and not properly aligned, bandwidth throughput drops. For instance, increasing image width by one pixel using an image of 5121×3200 pixels, gives us a peak throughput of 3.9 Gpixel/s which is roughly half of the throughput we got before. Using our framework allows to pad images and changes the kernel source to take padding into account. The amount of padding required for best performance depends on the underlying hardware. For the used graphics hardware, best memory throughput can be achieved when the image is padded to a multiple of the memory transaction size that can be handled by the GPU in one transaction. This size can be 32-, 64-, and 128-byte segments of aligned memory. Doing so improves the peak throughput as shown in Fig. 2 for an image of 5121×3200 pixels with image lines padded to the different memory transaction sizes. The peak throughput of 6.4 Gpixel/s is achieved for aligning to 256-bytes, which is double the maximum transaction size.

3.2 OpenCV Library

One widely used library for image processing is the *Open Source Computer Vision* (OpenCV) library [9]. Image processing algorithms are optimized in OpenCV to make

[4] The generated OpenCL code is slightly slower than the generated CUDA code, which we attribute to the relative immaturity of the OpenCL implementation.



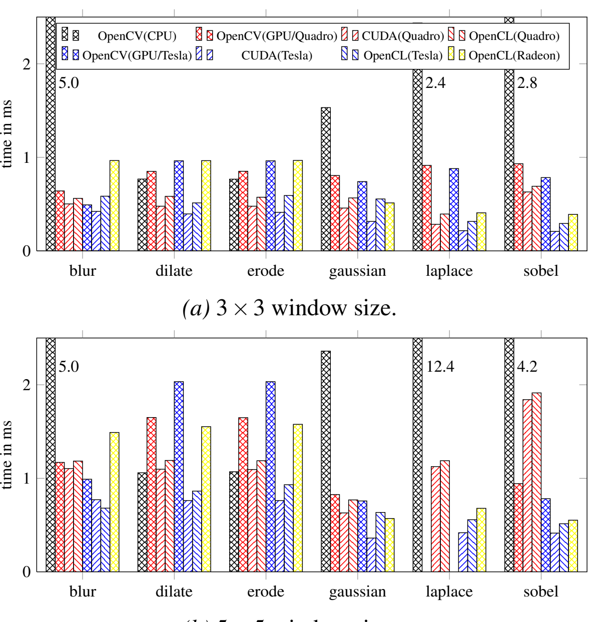
(a) CUDA. *(b)* OpenCL.

Fig. 2: Throughput of the generated CUDA C sources in Gpixel/s for the vertical mean filter on an image of 5121×3200 pixels with padding. The generated CUDA C and OpenCL source pads the image width to a multiple of 32-, 64-, 128-, 256-, or 512-bytes.

use of the SIMD units and multiple cores of modern processors. Beginning with version 2.2, selected algorithms (mostly convolution kernels) can also be executed on the GPU. Instead of implementing these kernels from scratch, OpenCV relies on the NVIDIA Performance Primitives (NPP) library. To compare the performance of code generated by our framework to such state-of-the-art approaches, we used the framework to implement all six convolution kernels from OpenCV that utilize NVIDIA GPUs. These kernels mostly support the 8-bit *unsigned char* type and the 3×3 and 5×5 window dimensions, which we use for evaluation. (For example, there is no 5×5 GPU implementation of the laplace convolution filter.) However, we can also generate code for other configurations with only minor modifications to the high-level description as for the 5×5 laplace convolution filter.

Figure 3 shows the execution times of the OpenCV implementations on a CPU (Core 2 Quad @3.00 GHz) and three GPUs: NVIDIA's Quadro FX 5800 and Tesla C2050 and AMD's Radeon HD 5870. For the NVIDIA cards, the OpenCV implementation and CUDA/OpenCL code generated by our framework are compared, while on the AMD card only generated OpenCL code is available. Generated code is as fast as OpenCV code (actually, faster in most cases). With larger filter window size also execution times increase. Again, the generated CUDA code is slightly faster than the OpenCL code. The GPU implementation of OpenCV relies on NPP, resulting in longer execution times. While our DSL approach generates GPU code from a high-level representation of the desired convolution kernel, the OpenCV library and NPP[5] provide more general implementations that are not optimized for the selected convolution kernel properties like the kernel size. The performance of the vectorized OpenCV code varies considerably. For some convolution kernels, their CPU implementation is almost as fast as our generated GPU code (e. g., for dilate and erode); for most kernels, however, their CPU implementation is an order of magnitude slower than generated GPU code (e. g., for blur, laplace, and gaussian). One big advantage of our framework is that we can

[5] The NPP source code is not available for detailed analysis.



(a) 3×3 window size.

(b) 5×5 window size.

Fig. 3: Comparison of the execution time of convolution kernels from OpenCV and our framework for an image of 1024×1024 pixels on a Quadro FX 5800, Tesla C2050, and Radeon HD 5870. The results for a window size of 3×3 is shown in (a) and for a window size of 5×5 in (b).

generate code for any pixel data type, while the OpenCV implementations are mostly restricted to *unsigned char*.

4 Future Work

The framework presented in this paper allows abstract description of algorithms for image processing which is translated and transformed into device-dependent, optimized source codes. While this works for simple kernels and convolution kernels, we are planning to extend our current framework to provide better support for a broader range of image processing specific features and applications.

The current version of our framework does not support border handling for image processing. The user has to specify border handling in the high-level algorithm description. Instead, our compiler can generate border handling support for images, like clamping to the last valid value, repeating the values beyond the border, mirroring the values at the border, or using a constant value.

Currently, the access/execute metadata is mostly implicit: we assume that the iteration space is parallel in all dimensions and has a 1:1 mapping to work-items (threads), and that the memory access pattern is obvious from the kernel code. In the vertical mean filter example, we use the offset specification to realize a 1:N mapping. More elegant, native support for such mappings allow not only more concise code, but also optimizations on the generated code.

The configuration for a kernel can be specified by a user or determined by the framework. Currently, our framework falls back to a default configuration of 128×1, which is generally suboptimal. However, the compiler can detect a suitable configuration for a kernel and use this setting.

Often, to reduce overheads for dispatching kernels and communicating intermediate data between them, the programmers *manually* bundle several kernels (related by data flow, not necessarily by the application logic!) into a single kernel. Since we have AST information for kernel functions, we can perform kernel fusion and other optimizations automatically if they are allowed by iteration space specifications and data dependences.

5 Conclusion

In this paper, we introduced a performance-portable framework for image processing. Our framework provides C++ classes that allow to describe image processing kernels based on decoupled access/execute metadata, which allows programmers to concentrate on developing algorithms and applications, rather than on mapping them to the target hardware.

The framework performs source-to-source translation of kernels expressed in high-level framework-specific C++ classes into low-level CUDA C and OpenCL code with effective device-dependent optimizations such as global memory padding for memory coalescing and optimal memory bandwidth utilization. Our source-to-source compiler is based on Clang and creates AST for kernel functions, which leaves room for future intra- and inter-kernel optimizations.

Our experiments show that code generated from our abstract description is as fast as hand-optimized and -tuned CUDA code for vertical mean filtering. While the performance for images that lead to misaligned memory layouts decreases almost by 50 %, our compiler pads the memory layout so that almost no performance penalty can be observed. In terms of lines of code, our concise high-level description requires only one tenth of the hand-written CUDA implementation. Supporting different backends, our source-to-source compiler produces CUDA C and OpenCL code that is faster than the OpenCV/NPP implementation for the available OpenCV convolution kernels that run on the GPU.

References

1. Clang: Clang: A C Language Family Frontend for LLVM. http://clang.llvm.org (2007–2011)
2. Cornwall, J., Howes, L., Kelly, P., Parsonage, P., Nicoletti, B.: High-Performance SIMT Code Generation in an Active Visual Effects Library. In: Proceedings of the 6th ACM Conference on Computing Frontiers. pp. 175–184. ACM (2009)
3. Du, P., Weber, R., Luszczek, P., Tomov, S., Peterson, G., Dongarra, J.: From CUDA to OpenCL: Towards a Performance-portable Solution for Multi-platform GPU Programming. Tech. rep. (2010)
4. Howes, L., Lokhmotov, A., Donaldson, A., Kelly, P.: Towards Metaprogramming for Parallel Systems on a Chip. In: Euro-Par 2009–Parallel Processing Workshops. pp. 36–45. Springer (2010)
5. Lin, C., Snyder, L.: Principles of Parallel Programming. Addison-Wesley Publishing Company, USA (2008)
6. NVIDIA: CUDA. http://www.nvidia.com/cuda (2006–2011)
7. Ryoo, S., Rodrigues, C., Stone, S., Stratton, J., Ueng, S., Baghsorkhi, S., Hwu, W.: Program Optimization Carving for GPU Computing. Journal of Parallel and Distributed Computing 68(10), 1389–1401 (2008)
8. The Khronos Group: OpenCL. http://www.khronos.org/opencl (2008–2011)
9. Willow Garage: Open Source Computer Vision (OpenCV). http://opencv.willowgarage.com/wiki (1999–2011)

A Greedy Heuristic Approximation Scheduling Algorithm for 3D Multicore Processors*

Thomas Canhao Xu, Pasi Liljeberg, and Hannu Tenhunen

Turku Center for Computer Science, Joukahaisenkatu 3-5 B, 20520, Turku, Finland
Department of Information Technology, University of Turku, 20014, Turku, Finland
{canxu, pasi.liljeberg, hannu.tenhunen}@utu.fi

Abstract. In this paper, we propose a greedy heuristic approximation scheduling algorithm for future multicore processors. It is expected that hundreds of cores will be integrated on a single chip, known as a Chip Multiprocessor (CMP). To reduce on-chip communication delay, 3D integration with Through Silicon Vias (TSVs) is introduced to replace the 2D counterpart. Multiple functional layers can be stacked in a 3D CMP. However, operating system process scheduling, one of the most important design issues for CMP systems, has not been well addressed for such a system. We define a model for future 3D CMPs, based on which a scheduling algorithm is proposed to reduce cache access latencies and the delay of inter process communications (IPC). We explore different scheduling possibilities and discuss the advantages and disadvantages of our algorithm. We present benchmark results using a cycle accurate full system simulator based on realistic workloads. Experiments show that under two workloads, the execution times of our scheduling in two configurations (2 and 4 threads) are reduced by 15.58% and 8.13% respectively, compared with the other schedulings. Our study provides a guideline for designing scheduling algorithms for 3D multicore processors.

1 Introduction

The number of circuits integrated on a chip have been increasing continuously which leads to an exponential rise in the complexity of their interaction. Traditional digital system design methods, e.g. bus-based System-on-Chip (SoC) will encounter communication bottlenecks. To address these problems, Network-on-Chip (NoC) was proposed as a promising communication platform solution for future multicore systems [1]. Network communication methodologies are brought into on-chip communication. Figure 1 shows a NoC with 4×4 mesh (16 nodes). The underlying network is comprised of network links and routers (R), each of which is connected to a processing element (PE) via a network interface (NI). The basic architectural unit of a NoC is a tile/node (N) which consists of a router, its attached NI and PE, and the corresponding links. Communication among PEs is achieved via network packets. Intel [1] has demonstrated an 80 tile,

* This work is supported by Academy of Finland and Nokia Foundation.

[1] Intel is a trademark or registered trademark of Intel or its subsidiaries. Other names and brands may be claimed as the property of others.

100M transistor, 275mm^2 2D NoC under 65nm technology [2]. Recently, an experimental microprocessor containing 48 cores (x86) on a chip has been created, using 4×6 2D mesh topology with 2 cores per tile [2].

Traditional 2D chip interconnection will result long global wire lengths, causing a high delay, high power consumption and low performance [3]. Besides 2D chips have larger die size in multiprocessor implementations. The 3D integration has the potential to increase device density, providing shorter wire lengths and faster on-chip communication compared with the 2D integration. Traditional stacking technologies such as System-in-Package (SiP) and Package-on-Package (PoP) have been integrated into manufacturing technology. Recent researches have focused on TSV [4]. TSV is a viable solution in building 3D chips by stacking IC layers together using vertical interconnects. These interconnects are formed through the silicon die to enable communication among layers. Layers with different functionalities can be implemented in a 3D chip. The manufacturing process of the TSV is complex and expensive [4], therefore finding an optimal number and placement of TSVs is critical. It is presented that, the balance between performance and manufacturing cost is essential in designing a 3D chip [5].

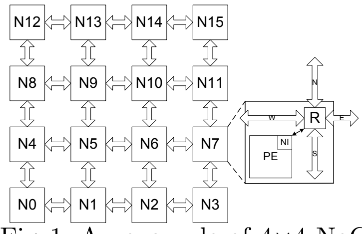


Fig. 1: An example of 4×4 NoC using mesh topology.

With limited resources between layers, it is obvious that better or even optimal efficiency can be achieved through appropriate scheduling of multi-threaded tasks in large scale 3D multicore processors. The design of operating system schedulers is one of the most important issues for CMPs. Several multiprocessor scheduling policies such as round robin, co-scheduling and dynamic partitioning have been studied and compared in [6]. However, these policies are designed mainly for the conventional shared bus based communication architecture. Many heuristic-based scheduling methods have been proposed [7, 8]. These methods are based on different assumptions, e.g. the prior knowledge of the tasks and execution time of each task in a program, presented as a directed acyclic graph. Hypercube scheduling has been proposed for off-chip systems [9]. Hypercube systems, usually based on Non-Uniform Memory Access (NUMA) or cache coherent NUMA architectures [10], are different from CMPs. Task scheduling for 2D NoC platforms is studied in [11] and [12]. The impact of limited resources between layers is not considered in these papers.

In our paper, we propose and discuss a novel greedy heuristic approximation scheduler for TSV constrained 3D multicore processors which aims to reduce the average network latency between caches and processing cores. With the decrease of the latencies, lower power consumption and higher performance can be achieved. To confirm our theory, we model and analyze a 64-core, 2-layer NoC with 8×8 meshes, present the performance of an application with different allocation strategies using a full system simulator.

2 3D NoC with Through Silicon Via Constraints

A modern multi-core processor is composed of several parts, e.g. processor core, shared last level cache, I/O and memory controller. Processor core and shared cache consume over 80% of the die area [13]. The total area of Sun SPARC chip is 396mm^2 with 65nm fabrication technology. Each core has an area of 14mm^2, thus with 16 cores the total area of cores is 224mm^2 (56.6%). Shared caches and other components occupy 172mm^2 (43.4%). As explained above, nearly half of the die area is devoted to cores and the other half is devoted to shared caches and other circuits. A natural way of applying 3D integration is to partition all the processors to one layer and other components to the other layer.

2.1 Processors and Caches

There is a significant concern for thermal hot-spots brought by packing layers vertically. It is expectable that since the processors consume overwhelming majority of power in a chip, stacking multiple processor layers would be unwise for heat dissipation. According to [5], heat dissipation is a major problem by stacking multiple processor layers even if processors are interlaced vertically. Therefore, in consideration of heat dissipation of current CMP, the processor layer should be on top of the chip.

In our paper, based on the above analysis, we use a 3D multicore processor model of two layers. The top layer is an 8×8 mesh of 64 Sun SPARC cores. Each core, scaled to 32nm technology, has an area of 3.4mm^2. We simulate the characteristics of a 64MB, 64 banks, 64-bit line size, 4-way associative, 32nm cache by CACTI [14]. Results show that the total area of cache banks is 204.33mm^2. Each cache bank, including data and tag, occupies 3.2mm^2. The cache layer has an 8×8 mesh of cache banks. Routers are quite small compared with processors and caches, e.g. we calculate a 7-port 3D router to be only 0.096mm^2 under 32nm. The total area of the processor is supposed to be below 300mm^2. Figure 2 shows a model with two layers and 16 processors only.

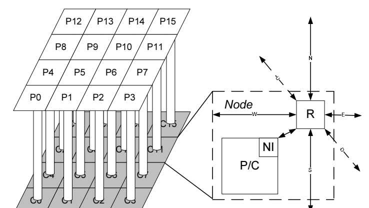


Fig. 2: A 3D NoC with one processor layer (upper) and one cache layer (lower), layers are fully connected by TSVs.

2.2 Constraints of the Through Silicon Via

TSV is the most promising solution for building 3D chips. There are several types of TSVs, e.g. data signal transmission, control signal transmission, power distribution and thermal dissipation. In our paper, a pillar is defined as a bunch of TSVs, including TSVs for data, control and power distribution. On the assumption that the power supply voltage is 1V, a practical aspect ratio for TSVs is between 10:1 to 5:1, in which signal TSVs are dominant ones [15]. As it is shown in Figure 1, routers in the 2D NoC have five ports to connect to five directions, namely, North, East, West, South and Local PE. For the vertical communication between different layers, routers in a generic 3D NoC model have two more ports and the corresponding virtual channels, buffers and crossbars to connect to the

Up and Down pillars (Figure 2). It is noteworthy that routers in our 3D NoC require less than seven ports, e.g. router of P12 in Figure 2 has only four ports (East, South, Local PE and Down).

It is obvious that the maximum performance can be achieved by full layer-layer connection, e.g. all routers are connected with up/down routers by pillars. However, as the number of tiles grow, it might not be practical to assume that each tile will be connected with corresponding TSVs because of the manufacturing cost and chip area. Assuming that a flit in a NoC is 128 bits, full layer-layer connection for an 8×8 NoC would require $128 \times 8 \times 8 = 8,192$ TSVs for parallel data signals. Other TSVs are required for power, thermal and control. Several researches have shown that [4, 16], TSV processing cost is the dominating cost for a 3D wafer. It is cheaper to manufacture a 3D chip with a fewer pillars between layers, in this case, multiple nodes have to share a pillar, high congestion could be created on a pillar, leading to communication bottlenecks. In [5], the placement of pillars is studied, an optimal placement of TSVs for an 8×8 mesh with 16 pillars is presented to minimize traffic contention between layers (Figure 3). The overall performance and total number of TSVs are 92% and 20% compared with full layer-layer connection respectively, achieving a good balance between performance and manufacturing cost.

56	57	58	59	60	61	62	63
48	49	50	51	52	53	54	55
40	41	42	43	44	45	46	47
32	33	34	35	36	37	38	39
24	25	26	27	28	29	30	31
16	17	18	19	20	21	22	23
8	9	10	11	12	13	14	15
0	1	2	3	4	5	6	7

Fig. 3: Gray nodes are attached with a pillar, number means node sequence.

Assuming X-Y-Z deterministic routing, Equation 1 shows the access time (latency) required for a core-cache communication. The latency involves in-tile links (Between NI and PE, L_{Link_delay1}), router (L_{Router_delay}), tile-tile links (L_{Link_delay2}), the number of hops required to reach the destination (n_{hop}) and the delay caused by TSV (L_{TSV_delay}). Since the delays of link, router and TSV are fixed, hop count is the most important metric in determining latency. Figure 4 shows the average hop counts required for a core accessing the shared cache nodes (AHPC). Obviously, without proper schedule, the communication overhead can be an obstacle. For example, nodes in corners of the NoC have much higher AHPC than nodes in the center. However, nodes directly connected with a pillar usually have lower AHPC, sometimes even lower than inner nodes, e.g. the AHPC for the node 38 is 5.75, lower than 6.75 of the node 37. Scheduling a task to the node 38 is therefore preferable than 37, since the average delay to the shared caches is lower.

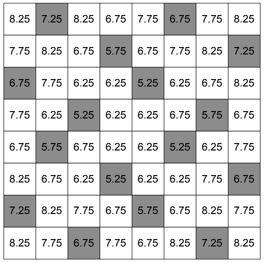


Fig. 4: Gray nodes are attached with a pillar, number means average hop counts to all cache nodes.

$$L_{CoreCache} = 2 \times L_{Link_delay1} + (n_{hop} + 1) \times L_{Router_delay} + n_{hop} \times L_{Link_delay2} + L_{TSV_delay} \quad (1)$$

3 The Scheduling Algorithm

Our proposed scheduling algorithm takes into consideration of on-chip topology and TSV placement, scheduling decisions are made based on these information. The aim of the algorithm is to minimize average network latency of the system, which is an important factor of system performance. We use a 3D NoC model as described below.

Definition 1 *A 3D NoC $N(P(X,Y),C(X,Y))$ consists of a layer of PE mesh $P(X,Y)$ (width X, length Y); and a layer of cache mesh $C(X,Y)$. Layers are connected by TSVs, only a quarter of nodes are connected (Figure 3).*

Definition 2 *Each node is denoted by a coordinate (x,y), where $0 \leq x \leq X-1$ and $0 \leq y \leq Y-1$.*

Definition 3 *The Manhattan Distance between two PEs $n_i(x_i,y_i)$ and $n_j(x_j,y_j)$ is MD(n_i,n_j), MD(n_i,n_j)$=|x_i-x_j|+|y_i-y_j|$. Two nodes in the same layer $n_1(x_1,y_1)$ and $n_2(x_2,y_2)$ are interconnected by a router and related link only if they are adjacent, e.g. MD(n_1,n_2)=1.*

Definition 4 *A task $T(n)$ with n threads requests the allocation of n cores.*

Definition 5 *n_{Free} is a sorted list of all unallocated nodes in P, such that: $AHPC_{nFree1} \leq AHPC_{nFree2} \leq AHPC_{nFree3} \leq \ldots AHPC_{nFreek}$.*

Definition 6 *$R(T(n))$ is a unallocated region in P with n cores for $T(n)$.*

To schedule a task efficiently, several metrics have to be considered, e.g. MD, AHPC and so on. Scheduling a task with only 1 thread is relatively easy. In this case, nodes 19, 29, 34 and 44 are considered in the first place, if they are not utilized by other applications. The reason is that, these four nodes have the lowest AHPC (5.25). However, as the requested number of threads grows, other metrics have to be included. For example, a 2-thread task can be scheduled to nodes 19 and 29. In this case, the Inter Process Communication (IPC) between threads will suffer higher delay, since the messages have to go through nodes 20 and 21 according to XY routing. Another problem is fragmentation. Non-contiguous allocation of cores in a dynamic system can cause degradation of overall system performance. The 2-thread task can be scheduled to nodes 19 and 20 as well. Despite the fact that the AHPC is increased by 1 for node 20 compared with node 29, the adjacent allocation will alleviate IPC bottleneck, and reduce fragmentation. We introduce Average Core-access Time (ACT), which is defined as the number of nodes a message has to go through from a PE to other PEs, $\forall i,j \in P$.

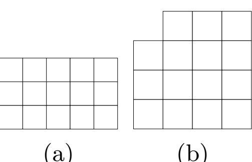
$$ACT = \frac{\sum MD(n_i,n_j)}{n} \tag{2}$$

Such that: $\forall i \neq j \in P$ and $n_i \neq n_j$

According to the equation, the ACT is 3 and 1 for nodes 19/29 and 19/20, respectively. The delay for a core-core communication is shown in Equation 3. Obviously, allocation 19/29 will incur much higher router delay and delay of tile-tile links, comparing with allocation 19/20. It is noteworthy that a core-core communication is a intra-layer communication, while a core-cache communication is a inter-layer communication.

$$L_{CoreCore} = 2 \times L_{Link_delay1} + (n_{hop}+1) \times L_{Router_delay} + n_{hop} \times L_{Link_delay2} \quad (3)$$

For a rectangular core allocation with $A \times B$ nodes, according to [17], ACT can be calculated in an easier way (Equation 4). For example, 4×4 and 2×8 are possible rectangular core allocations for a task with 16 threads. However, the value of ACT in 4×4 is smaller than in 2×8 (2.5 and 3.125). In consideration of ACT, an allocation shape have a lower ACT number if it is closer to a square. Figure 5a and 5b show two core allocation schemes for a task with 15 threads. In Figure 5b, the number of ACT is lower than in Figure 5a (2.4177 and 2.4888 respectively).



(a) (b)

Fig. 5: Comparison of two core allocation schemes for 15 threads.

$$ACT = \frac{A+B}{3} \times \left(1 - \frac{1}{A \times B}\right) \quad (4)$$

A scheduling algorithm should have a low computation complexity and should deliver an optimal or near-optimal results. This is due to the scheduling has to be solved online, and the time for solving the scheduling is a part of the overall system response time. It is clear that we should not try to solve the scheduling problem optimally, in case the computation complexity is too high. Given a task with n executing threads, we define the problem as determining the near-optimal core allocation for the task by selecting a region containing of n cores. The pseudo code of the algorithm is shown in Algorithm 1.

Algorithm 1 The Greedy Heuristic Approximation Scheduling Algorithm

Input: A mesh based NoC N with TSV constrains, a task with n threads
Output: An allocated region R, containing n processors

```
1  Pop the first node as an initial node u_0 from n_Free, and push u_0 to R
2  n_MD := n_Free sorted as MD(u_0, n_Free)
3  while n_MD is not empty do
4  |   Pick a node u_n(x_i, y_i) from n_MD with smaller AHPC
   |   if several nodes with same AHPC then
5  |   |   pick a node u_n(x_i, y_i) with smaller ACT in the result region
6  |   end
7  |   if several nodes with same ACT then
8  |   |   pick a node u_n(x_i, y_i) which x_i -> X/2 and y_i -> Y/2
9  |   end
10 |   Pop u_n(x_i, y_i) from n_MD, and push u_n to R
11 end
```

Line 1 sets the starting node of the algorithm, which is the one with the lowest AHPC. A list n_{MD} contains nodes sorted based on MD from the starting node. The adjacent nodes are always considered first, in terms of AHPC. ACT will be calculated, in case several nodes are with the same AHPC. If ACTs for the allocation strategies are still the same, a node closer to the center of the network will be selected (considering the statistical variance of the coordinates of two nodes, Equation 5). This is due to the fact that, nodes in the center usually have lower AHPC than nodes in the border, following steps may have better results from this heuristic.

$$Var(n) = \frac{1}{2} \times [(x_i - \frac{X}{2})^2 + (x_j - \frac{Y}{2})^2] \quad (5)$$

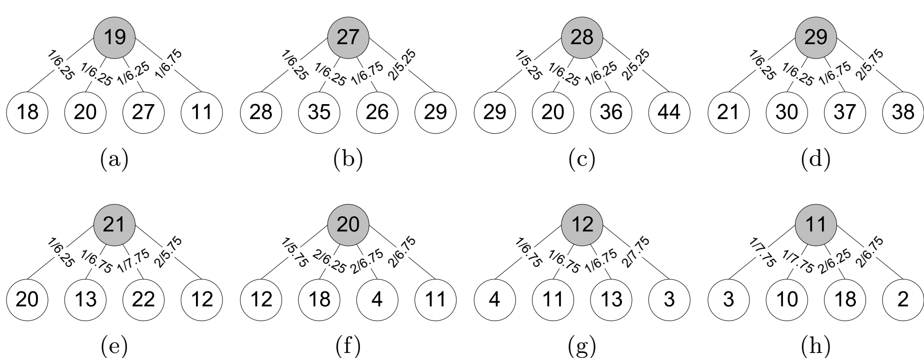


Fig. 6: The node selection steps for the algorithm.

We analyze an example of the algorithm. Figures 6a to 6h shows the steps for node selection, starting from node 19. The number between two nodes n_i and n_j means MD(n_i,n_j) and AHPC(n_j). Note that we only show 4 child nodes in these figures. The actual list n_{MD} and n_{Free} may contain more nodes. As illustrated in Figure 6a, node 19 has 4 adjacent nodes and 3 of them are with the same AHPC and ACT. However, in terms of distance to the center, node 27 is selected ($Var(27) < Var(20) < Var(18)$). The selection of the next node follows the similar rule: same AHPC, same ACT, same variance. In this case we choose node 28, having a smaller node number than node 35. Figure 6c demonstrates that, node 29 is selected due to its lowest MD and AHPC. The next step involves different ACTs: both node 21 and node 30 have the lowest AHPC, however the ACTs for the two nodes are different (2 for node 30, and 1.8 for node 21). Node 20 and 12 are selected as the sixth and seventh node, respectively, due to their lowest AHPC. The next node (11) is picked out, on account of its lower ACT than the others. It is noteworthy that the aforementioned greedy heuristic approximation algorithm generates near-optimal scheduling solution in most cases. However, in our algorithm we put adjacent nodes as the first priority, the AHPC and ACT are considered next. This strategy may generate non-optimal scheduling for certain applications.

Take a 4-thread application for example. As shown in Figure 7, the algorithm will choose nodes 19, 27, 28 and 29 for allocation.

An IPC-intensive application may suffer from the long distance communication of node 19 and 29. In this case, node 20 is a better choice than 29 since the ACT is lower. Despite our goal is to find a near-optimal scheduling using MD, AHPC and ACT, the weight of these metrics should be considered as well. Different applications have their own profile: some have higher demand of caches, some have higher volume of IPC. It is difficult to determine the behavior of an application automatically beforehand, since there are millions of them and the number is still increasing. One feasible way is to add an interface between the application and the OS, the application will tell the OS its behavior. Another way is to add a low overhead profiling module inside the OS. Program access patterns are traced dynamically, and possibly migrated for better allocations.

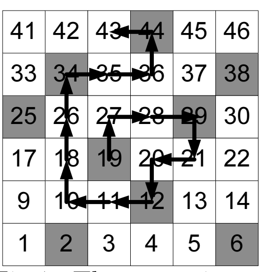


Fig. 7: The execution of our algorithm, starting from node 19 and selected 16 nodes.

4 Experimental Evaluation

4.1 Experiment Setup and Application

The simulation platform is based on a cycle-accurate NoC simulator which is able to produce detailed evaluation results. The platform models the routers and links accurately. The router includes a routing computation unit, a virtual channel allocator, a switch allocator, a crossbar switch and 4 input buffers. Deterministic XYZ routing algorithm has been selected to avoid deadlocks. We use a 64-core multiprocessor which models a single-chip CMP for our experiments. The 3D architecture in this paper has two layers; the first layer contains PEs (each running at 2GHz with a private L1 cache, split I+D, each 16KB, 4-way associative, 64-bit line, 3-cycle), the second layer consists of shared L2 caches (unified 64 banks, each 1MB, 64-bit line, 6-cycle). The simulations are run on Solaris 9 based on UltraSPARCIII+ instruction set in-order issue structure. The simulated memory/cache architecture mimics Static Non-Uniform Cache Architecture. A two-level directory cache coherence protocol called MOESI, based on MESI, has been implemented in our memory hierarchy in which each L2 bank has its own directory. We use Simics [18] as our full system simulation platform.

We select FFT [19] and Radix [20] as experiment applications. The FFT algorithm is a one-dimensional, radix-n, six-step algorithm optimized to minimize IPC. The communication between processors only take place at the last stage of the execution. However the network traffic and cache miss rate are very high. The Radix sort algorithm assigns each processor a part of the sorting keys. For every iteration in the algorithm, a permutation for the keys is required to create a new array for the next iteration. This will incur all-to-all communication among processes. Hence Radix represents an application with high IPC.

4.2 Result Analysis

We evaluate performance in terms of Average Network Latency (ANL), Execution Time (ET) and Cache Hit Latencies (CHL). ANL represents the average number of cycles required for the transmission of all messages. The number of

required cycles for each message is calculated from the injection of the message header into the network at the source node, to the reception of the tail flit at the destination node. Under the same configuration and workload, lower values are favorable. We analyze two core allocations for a 2-thread task: *T2-1* is from our algorithm, which contains nodes 19 and 27. It has lowest ACT values, however the AHPC is not optimal. *T2-2* is an alternative allocation, which contains nodes 19 and 29. In this case, the AHPC is minimized. The *T4-1*, *T4-2* and *T4-3* are three core allocations for a 4-thread task: *T4-2* contains nodes 19, 20, 27 and 28, represents lowest ACT; *T4-3* contains nodes 19, 29, 34 and 44, represents lowest AHPC. Our algorithm selects *T4-1*, it has neither lowest ACT nor AHPC numbers. However we believe it could be a good balance for the two metrics.

The results are illustrated in Figure 8. The core allocation of our scheduling algorithm for 2 threads outperforms the other in terms of ANL. The improvement is more notable in 2-thread FFT and Radix, with 9.26% and 11.77% reduced latency, respectively, compared with the *T2-2* allocation. This is primarily due to the reduced communication overhead between two PEs. We note that the reduced AHPC in *T2-2* failed to compensate the increasing ACT, in terms of ANL. The CHL in *T2-2* directly reflects the reduced AHPC. However, the average runtime of two applications show that, our algorithm spends 15.58% shorter time than *T2-2*. Considering a 4-thread task, we note that both ACT and AHPC play important roles in overall performance. For example, despite the fact that *T4-2* has lowest ACT, the ANL for two applications is 3.76% higher than in our algorithm. This leads to a higher ET as well. Allocation *T4-3* performs better than our scheduling in the 4-thread FFT. This is because of, in FFT, the communication between threads only happens at the last stage of the execution. In this case, we observe that the trade-off for ACT is worthy. However, applications that heavily rely on IPC, e.g. Radix, will suffer from the *T4-3*. The ET of *T4-3* is 24.24% longer than in *T4-1*.

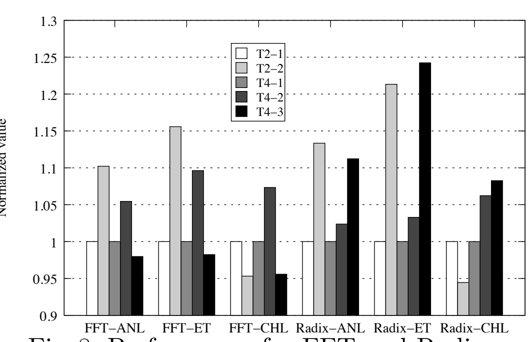


Fig. 8: Performance for FFT and Radix.

5 Conclusion and Future Work

In this paper, we studied the problem of process scheduling for future 3D multicore processors. A model for NoC-based 3D CMP was defined. We analyzed process scheduling in terms of average hop counts for core-cache accesses (AHPC) and average core access time (ACT) in 3D CMPs with constraints of inter-layer connections. A greedy heuristic approximation algorithm was proposed to reduce overall on-chip communication latencies and improve performance. Results have shown that, with proper scheduling, performance improved significantly in most cases. The impact of ACT and AHPC was discussed. The results of this paper

give a guideline in designing schedulers for future 3D CMPs. Our next step is to evaluate more applications with different access profiles and number of threads. The weight of AHPC and ACT will be analyzed and compared, and the trade-off for finding the best allocation strategy will be studied.

References

1. Dally, W.J., Towles, B.: Route packets, not wires: on-chip inteconnection networks. In: Proceedings of the 38th conference on Design automation. (June 2001) 684–689
2. Intel: Intel research areas on microarchitecture (May 2011) http://techresearch.intel.com/projecthome.aspx?ResearchAreaId=11.
3. Sylvester, D., Keutzer, K.: Getting to the bottom of deep submicron. In: ICCAD 98. (Nov 1998) 203–211
4. Velenis, D., Stucchi, M., Marinissen, E., Swinnen, B., Beyne, E.: Impact of 3d design choices on manufacturing cost. In: IEEE 3DIC 2009. (Sept. 2009) 1–5
5. Xu, T.C., Liljeberg, P., Tenhunen, H.: Optimal number and placement of through silicon vias in 3d network-on-chip. In: Proc. of the 14th DDECS, IEEE (2011)
6. Leutenegger, S.T., Vernon, M.K.: The performance of multiprogrammed multiprocessor scheduling algorithms. In: Proc. of the SIGMETRICS. (April 1990) 226–236
7. Chen, C., Lee, C., Hou, E.: Efficient scheduling algorithms for robot inverse dynamics computation on a multiprocessor system. Systems, Man and Cybernetics, IEEE Transactions on **18**(5) (1988) 729 –743
8. Hakem, M., Butelle, F.: Dynamic critical path scheduling parallel programs onto multiprocessors. In: In Proceedings of 19th IEEE IPDPS. (2005) 203b
9. Sharma, D.D., Pradhan, D.K.: Processor allocation in hypercube multicomputers: Fast and efficient strategies for cubic and noncubic allocation. IEEE Transactions on parallel and distributed systems **6**(10) (October 1995) 1108–1123
10. Laudon, J., Lenoski, D.: The sgi origin: a ccnuma highly scalable server. In: Proc. of the 24th international symposium on Computer architecture. (June 1997) 241–251
11. Chen, Y.J., Yang, C.L., Chang, Y.S.: An architectural co-synthesis algorithm for energy-aware network-on-chip design. J. Syst. Archit. **55**(5-6) (2009) 299–309
12. Hu, J., Marculescu, R.: Energy-aware communication and task scheduling for network-on-chip architectures under real-time constraints. In: DATE '04, Washington, DC, USA, IEEE Computer Society (2004) 10234
13. IBM: Ibm power 7 processor. In: Hot chips 2009. (August 2009)
14. Shyamkumar, T., Naveen, M., Ho, A.J., P., J.N.: Cacti 5.1. Technical Report HPL-2008-20, HP Labs
15. Association, S.I.: The international technology roadmap for semiconductors (itrs) (2007) http://www.itrs.net/Links/2007ITRS/Home2007.htm.
16. Lau, J.H.: Tsv manufacturing yield and hidden costs for 3d ic integration. In: Proc. of the 60th ECTC. (2010) 1031 –1042
17. Lei, T., Kumar, S.: A two-step genetic algorithm for mapping task graphs to a network on chip architecture. In: DSD, 2003. (sep. 2003) 180 – 187
18. Magnusson, P., Christensson, M., Eskilson, J., Forsgren, D., Hallberg, G., Hogberg, J., Larsson, F., Moestedt, A., Werner, B.: Simics: A full system simulation platform. Computer **35**(2) (February 2002) 50–58
19. Bailey, D.H.: Ffts in external or hierarchical memory. The Journal of Supercomputing **4** (1990) 23–35 10.1007/BF00162341.
20. Blelloch, G.E., Leiserson, C.E., Maggs, B.M., Plaxton, C.G., Smith, S.J., Zagha, M.: A comparison of sorting algorithms for the connection machine cm-2. In: Proceedings of the 3rd SPAA, New York, NY, USA, ACM (1991) 3–16

the 5th Workshop on
Highly Parallel Processing
on a Chip