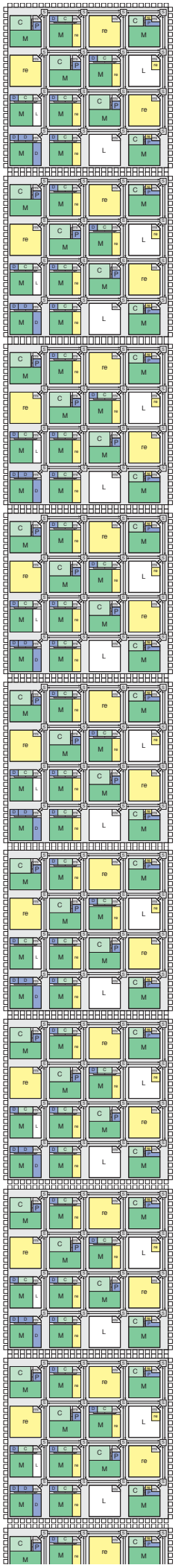
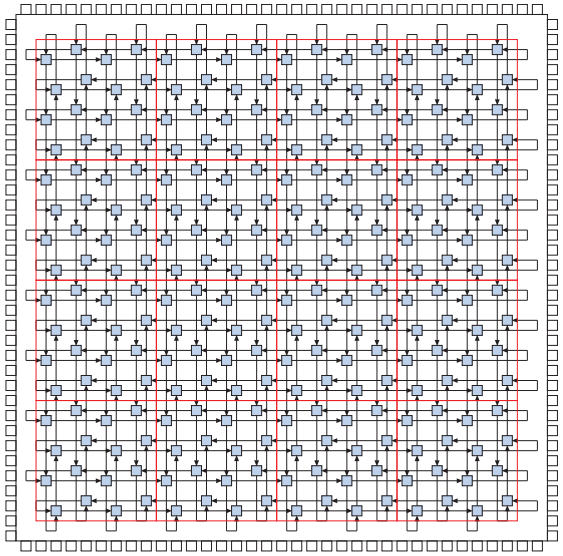
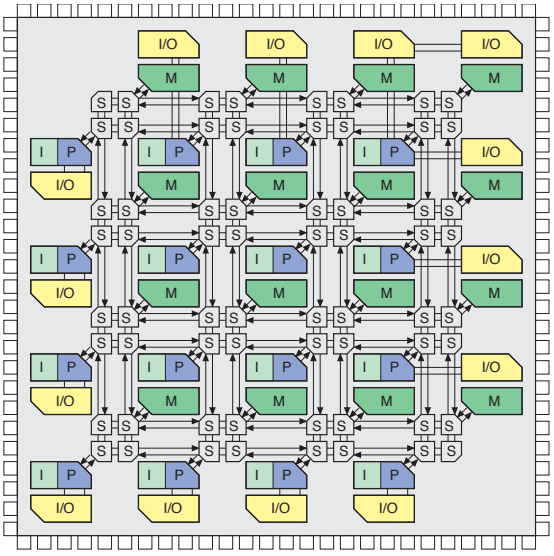






HPPC 2009

(Hand-out) Proceedings of the

the 3rd Workshop on
Highly Parallel Processing
on a Chip

August 25, 2009, Delft, The Netherlands
Organizers Martti Forsell and Jesper Larsson Träff

in conjunction with

the 15th International European Conference on
Parallel and Distributed Computing (Euro-Par)
August 25-28, 2009, Delft, The Netherlands

Sponsored by VTT NEC Euro-Par

(Hand-out) Proceedings of the

3rd Workshop on Highly Parallel Processing on a Chip

August 25, 2009, Delft, The Netherlands
http://www.hppc-workshop.org/

in conjunction with

the 15th International European Conference on Parallel and Distributed Computing (Euro-Par)
August 25-28, 2009, Delft, The Netherlands

August 2009
Handout editors: Martti Forsell and Jesper Larsson Träff
Printed in Finland and Germany

CONTENTS

FOREWORD

Technological developments are bringing parallel computing back into the limelight after some years of absence from the stage of main stream computing and computer science between the early 1990ties and early 2000s. The driving forces behind this return are mainly advances in VLSI technology: increasing transistor densities along with hot chips, leaky transistors, and slow wires make it unlikely that the increase in single processor performance can continue the exponential growth that has been sustained over the last 30 years. To satisfy the needs for application performance, major processor manufacturers are instead planning to double the number of processor cores per chip every second year (thus reinforcing the original formulation of Moore's law). We are therefore on the brink of entering a new era of highly parallel processing on a chip. However, many fundamental unresolved hardware and software issues remain that may make the transition slower and more painful than is optimistically expected from many sides. Among the most important such issues are convergence on an abstract architecture, programming model, and language to easily and efficiently realize the performance potential inherent in the technological developments.

This is the third 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 early in the year, and at the passing of the submission deadline we had received 18 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 at least 4 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. Being a single day event, we had room for accepting only 5 of the contributions, resulting in an acceptance ratio of about 28%. The 5 accepted contributions will be presented at the workshop today, together with two forward looking invited talks by Peter Hofstee and Ahmed Jerraya on the next 25 years of computer architecture and building a concurrency and software development and programming of multicore SOC. This year the workshop includes also a panel session on bringing together 5 distinguished panelists including Peter Hofstee, Ahmed Jerraya, Chris Jesshope, Uzi Vishkin, and Martti Forsell to discuss whether many-core processor vendors are on track to scalable machines that can be effectively programmed for parallelism by a broad group of users.

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 2009 conference. We also warmly thank our sponsors VTT, NEC and Euro-Par for the financial support which made it possible for us to invite Peter Hofstee and Ahmed Jerraya as well as the panelists, all 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 Delft, The Netherlands. We wish you all a productive and pleasant workshop.

HPPC organizers
Martti Forsell, VTT, Finland
Jesper Larsson Träff, NEC Europe, Germany

ORGANIZATION

Organized in conjuction with the 15th International European Conference on Parallel and Distributed Computing

WORKSHOP ORGANIZERS

Martti Forsell, VTT, Finland
Jesper Larsson Träff, NEC Laboratories Europe, NEC Europe Ltd, Germany

PROGRAM COMMITTEE

David Bader, Georgia Institute of Technology, USA
Gianfranco Bilardi, University of Padova, Italy
Marc Daumas, University of Perpignan Via Domitia, France
Martti Forsell, VTT, Finland
Peter Hofstee, IBM, USA
Chris Jesshope, University of Amsterdam, The Netherlands
Ben Juurlink, Technical University of Delft, The Netherlands
Jorg Keller, University of Hagen, Germany
Christoph Kessler, University of Linkoping, Sweden
Dominique Lavenier, IRISA - CNRS, France
Ville Leppanen, University of Turku, Finland
Radu Marculescu, Carnegie Mellon University, USA
Lasse Natvig, NTNU, Norway
Geppino Pucci, University of Padova, Italy
Jesper Larsson Traff, NEC Laboratories Europe, NEC Europe Ltd, Germany
Uzi Vishkin, University of Maryland, USA

SPONSORS

VTT, Finland	http://www.vtt.fi
NEC	http://www.it.neclab.eu/
Euro-Par	http://www.euro-par.org

PROGRAM

3rd Workshop on
Highly Parallel Processing on a Chip (HPPC 2009)

August 25, 2009, Delft, The Netherlands
http://www.hppc-workshop.org/

in conjunction with

the 15th International European Conference on Parallel and Distributed Computing (Euro-Par)
August 25-28, 2009, Delft, The Netherlands.

TUESDAY AUGUST 25, 2009

SESSION 1 - Multicore architectures

09:30-09:35 Opening remarks - *Jesper Larsson Träff and Martti Forsell, NEC Laboratories Europe, VTT*
09:35-10:35 Keynote - The next 25 years of computer architecture? - *Peter Hofstee, IBM Systems and Technology Group*
10:35-11:00 Distance Constrained Mapping to Support NoC Platforms based on Source Routing - *Rafael Tornero, Shashi Kumar, Saad Mubeen, Juan Manuel Orduña, University of Valencia, Jönköping University*

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

SESSION 2 - Programming

11:30-11:55 Parallel Variable-Length Encoding on GPGPUs - *Ana Balevic, University of Stuttgart*
11:55-12:20 Toward generative programming for parallel systems on a chip - *Lee Howles, Anton Lokhmotov, Alastair Donaldson, Paul Kelly, Imperial College London, University of Oxford*
12:20-12:45 Dynamic detection of uniform and affine vectors in GPGPU computations - *Sylvain Collange, David Defour, Yao Zhang, University of Perpigan*

12:45-14:30 -- Lunch --

SESSION 3 - Application-specific multicores

14:30-14:55 Automatic Calibration of Performance Models on Heterogeneous Multicore Architectures - *Cédric Augonnet, Samuel Thibault, Raymond Namyst, INRIA, LaBRI, University of Bordeaux*
14:55-15:55 Keynote - Software Development and Programming of Multicore SOC - *Ahmed Jerraya, CEA-LETI, MINATEC*

15:55-16:30 -- Break --

SESSION 4 - Panel

16:30-17:55 Panel - Are many-core computer vendors on track? - *Martti Forsell, Peter Hofstee, Ahmed Jerraya, Chris Jesshope, Uzi Vishkin, VTT, IBM Systems and Technology Group, CEA-LETI, University of Amsterdam, University of Maryland* - Moderator Jesper Larsson Träff, NEC Laboratories Europe
17:55-18:00 Closing notes - *Jesper Larsson Träff and Martti Forsell, NEC Europe, VTT*

KEYNOTE

The next 25 years of computer architecture?

Peter Hofstee, IBM Systems and Technology Group, USA

Abstract: This talk speculates on a technology-driven path computer architecture is likely to have to follow in order to continue to deliver application performance growth over the next 25 years in a cost- and power constrained environment. We try to take into account transistor physics, economic constraints, and discuss how one might go about programming systems that will look quite different from what we are used to today.

Bio: *H. Peter Hofstee is the IBM Chief Scientist for the Cell Broadband Engine processors used in systems from the Playstation 3 game console to the Roadrunner petaflop supercomputer. He has a masters (doctorandus) degree in theoretical physics from the Rijks Universiteit Groningen, and a PhD in computer science from Caltech. After two years on the faculty at Caltech, Peter joined the IBM Austin research laboratory in 1996 to work on the first GHz CMOS microprocessor. From 2001 to 2008 he worked on Cell processors and was the chief architect of the Synergistic Processor Element.*

Distance Constrained Mapping to Support NoC Platforms based on Source Routing*

Rafael Tornero[1], Shashi Kumar[2], Saad Mubeen[2], and Juan Manuel Orduña[1]

[1] Departamento de Informática, Universitat de València, Spain
{Rafael.Tornero,Juan.Orduna}@uv.es
[2] School of Engineering, Jönköping University, Sweden
{Shashi.Kumar,mems07musa}@jth.hj.se

Abstract. Efficient NoC is crucial for communication among processing elements in a highly parallel processing systems on chip. Mapping cores to slots in a NoC platform and designing efficient routing algorithms are two key problems in NoC design. Source routing offers major advantages over distributed routing especially for regular topology NoC platforms. But it suffers from a serious drawback of overhead since it requires whole communication path to be stored in every packet header. In this paper, we present a core mapping technique which helps to achieve a mapping with the constraint over the path length. We demonstrate the feasibility of reducing the path length to just 50% of the diameter. We also present a method to efficiently compute paths for source routing leading to good traffic distribution. Evaluation results show that performance degradation due to path length constraint is negligible at low as well at high communication traffic.

Key words: Network on Chip, Core Mapping, Routing Algorithms, Source Routing

1 Introduction

As Semi-conductor Technology advances, it becomes possible to integrate a large number of Intellectual Property (IP) cores, like DSPs, CPUs, Memories, etc, on a single chip to make products with complex and powerful functions. Efficient communication infrastructure is crucial for harnessing enormous computing power available on these Systems on Chip (SoCs). Network on Chip (NoC) is being considered as the most suitable candidate for this job [1].

Many design choices and aspects need to be considered for designing a SoC using NoC paradigm. These include: network topology selection, routing strategy selection and application mapping. Both application mapping and routing strategy have big impact on the performance of the application running on a NoC platform. The application mapping problem consist of three tasks: i) the

* This work has been jointly supported by the Spanish MEC and European Commission FEDER funds and the University of Valencia under grants Consolider Ingenio-2010 CSD2006-00046 and TIN2009-14475-C04-04 and V-SEGLES-PIE Program.

application is split into a set of communication tasks, generally represented as a task graph; ii) the tasks are assigned and scheduled on a set of IP cores selected from a library; iii) the IP cores have to be placed onto the network topology in such a way that the desired metrics of interest are optimized. The first two steps are not new since they have been extensively addressed in the area of hardware/software co-design and IP reuse [2] by the CAD community. The third step, called topological mapping, has recently been addressed by a few research groups [3], [4].

One way to classify the routing algorithms is by considering the component in the network where the routing decision to select the path is done. Under this consideration the routing algorithms are classified into source routing and distributed routing algorithms. In source routing algorithms, the path between each pair of communicating nodes is computed offline and stored at each source node. When a core needs to communicate with another core the encoded path information is put in the header of each packet. In distributed routing, the header only needs to carry the destination address and each router in the network has competence to make the routing decision based on the destination address.

Source routing has not been considered so far for NoCs due to its perceived underutilization of network bandwidth due to the requirement of large number of bits in the packet header to store path information. This conclusion may be valid perhaps for large dynamic networks where network size and topology are changing. But in a NoC with fixed and regular topology like mesh, the path information can be efficiently encoded with small number of bits. Saad et. al. [5] have made a good case for use of source routing for mesh topology NoCs. It can be easily shown [6] that two bits are sufficient to encode information about one hop in the path. Since the packet entering a router contains the pre-computed decision about the output port, the router design is significantly simplified. Also, since NoCs used in embedded systems are expected to be application specific, we can have a good profile of the communication traffic in the network [7]. This allows us to offline analyze the traffic and compute efficient paths according to the desired performance characteristics, like uniform traffic load distribution, reserved paths for guaranteed throughput etc.

Figure 1 shows an application, that has been assigned and scheduled on eight cores, topologically mapped on a 4x2 mesh. The Application Characterization Graph (APCG) of this application can be seen in Figure 1(a), where a node corresponds to a core and a directed edge corresponds to communication between two connected cores. APCG will be defined more formally in section 2. Assuming minimal routing, the maximum route length required is equal to the diameter of the topology. It means that, if the diameter was used for this example, the required path length would be 4 hops and therefore 10 bits would be required to code a route (see Figure 1(b)). However, it is possible to find a mapping in which the maximum distance between two communicating cores is much smaller than the diameter. Figure 1(c) shows such a mapping for the example in which the maximum distance is just two hops and only 6 bits are required for the path information.

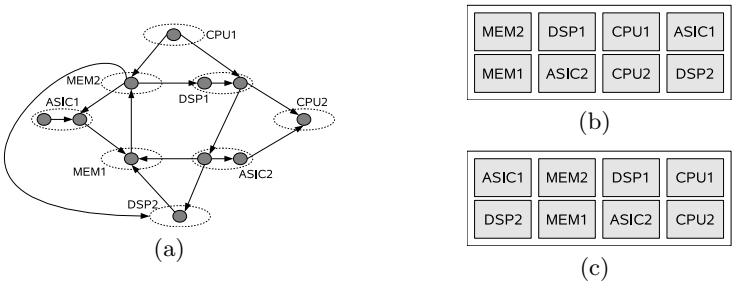


Fig. 1. Differents mappings of the same application. (a) the APCG, (b) A distance unconstrained mapping, (c) A 2-hops constrained mapping

Close compactness of mapping could lead to higher congestion in certain links. Since routers for source routing are relatively faster than routers for distributed routing, the above disadvantage will be adequately compensated [6].

Related Work

A large number of routing algorithms have been proposed in literature for NoCs. Most proposals fall in the category of distributed adaptive routing algorithms and provide partial adaptivity thus providing more than one path for most communicating pairs and at the same time avoiding possibility of deadlocks. In [7], Palesi et al. propose a methodology to compute deadlock free routing algorithms for application specific NoCs with the goal of maximizing the communication adaptivity.

Several works have been proposed in literature in the context of core mapping. Hu et al. present a branch and bound algorithm for mapping cores in a mesh topology NoC architecture that minimizes the total amount of power consumed in communications [8]. Murali et al. present a work to solve the mapping problem under bandwdith constraint with the aim of minimizing the communication delay by exploiting the possibility of splitting the traffic among various paths [3]. Hansson et al. present an unified single objective algorithm which couples path selection, mapping of cores, and channel time-slot allocation to minimize the network required to meet the constraints of the application [9]. Tornero et al. present a communication-aware topological mapping technique that, based on the experimental correlation of the network model with the actual network performance, avoids the need to experimentally evaluate each mapping explored [4]. In [10], Tornero et al. present a multi-objective strategy for concurrent mapping and routing for NoC. All the aforementioned works address the integration of mapping and routing concurrently but taking into account only the distributed routing functions.

Although source routing has been shown to be efficient for general networks [11], it had not been explored so far for NoC architectures. Recently Mubeen et.al [5] have made a strong case for source routing for small size mesh topology NoCs. The author, has demonstrated that source routing can have higher com-

munication performance than adaptive distributed routing [6]. In this paper we modify our earlier approach and tackle the integration of topological mapping for NoC platforms which use limited path length source routing for inter-core communication.

2 Problem Formulation

Simply stated, our goal is to find an arrangement of cores in tiles together with path selection such that the global communication cost is minimized and the maximum distance among communicating cores is within the given threshold. The value of the threshold comes from the fixed on-chip communication infrastructure (NoC) platform which uses source routing with an upper limit on the length of the path. Before formally defining the problem, we need to introduce the following definitions [8].

Definition 1 An Application Characterization Graph $APCG = G(C, A)$ is a directed graph, where each vertex $c_i \in C$ represents a selected IP core, and each directed arc $a \in A$ characterizes the communication process from core c_i to core c_j. For each communication $a \in A \wedge a = (c_i, c_j)$, the function $B(a)$ returns the bandwith requirements of a. This is minimum bandwith that should be allocated by the network in order to meet the performance constraint for communication a.

Definition 2 An Architecture Characterization Graph $ARCG = G(T, L)$ is a directed graph which models the network topology. Each vertex t_i represents a tile, and each directed arc l_{ij} represents the channel from tile t_i to tile t_j.

We must solve two problems: first, we have to find a mapping within the constraint of maximum distance allowed by the communication platform. The second problem to be solved is to compute efficient paths for all communicating pairs of cores such that there is no possibility of deadlock as well as the traffic is well balanced. We can formulate the first problem as follows. Given the APCG and the ARCG, that satisfy $|C| \leqslant |T|$, find a mapping function M from C to T which minimizes the mapping cost function M_c:

$$\min\{M_c = \sum_{\substack{\forall c_i, c_j \in C \\ a=(c_i,c_j)\in A}} B(a) * (dist(M(c_i), M(c_j)))^3\} \quad (1)$$

such that:

$$dist(M(c_i), M(c_j)) \leq Threshold \wedge a = (c_i, c_j) \in A\ . \quad (2)$$

In the equation 1, the second term of the summation is raised to power 3 with the aim of giving more importance to the distance in the search for a pseudo-optimal mapping. This value is a trade-off between the quality of the results and the computation time (the power 2 provides poor quality of results and power of 4 and higher ones are too much time consuming). Condition 2

guarantees that every pair of communicating cores should be mapped such that the Manhattan distance between them is less than the **threshold**. We assume that the underlying source routing uses only minimal distances. Nevertheless, this **threshold** cannot be smaller than the lower bound on the path length required for mapping APCG on ARCG. The lower bound in this context refers to a value such that any possible mapping will have at least one pair with distance more than or equal to the lower bound. For example, the lower bound for the APCG in our example is 2, since there is no possibility to find a mapping with distance 1. The APCG together with the ARCG can be analyzed in order to find the lower bound for the mapping.

In a 4x2 mesh topology, a node can be connected to maximum three other cores with a distance 1. A core can be connected to up to 6 cores if distance of 2 hops is allowed. There may not be any 2 distance constrained mapping available for an APCG with maximum out-degree 5. If L is the lower bound, then one can start by searching for a mapping with constraint equal to L. If one fails then one must repeat the process of finding a feasible mapping by using $L+1, L+2, \ldots$ and so on as the constraint.

Once the cores are mapped satisfiying (2), the second problem is to find a path for every communicating core pair C_i and C_j such that: the path length is equal to manhattan distance between C_i and C_j; there is no possibility of deadlock when some or all other core pairs communicate concurrently and the traffic load on all the links in the network is as balanced as possible.

3 The Distance Constrained Mapping Algorithm

We have modified our earlier mapping approach developed for NoC platforms using distributed routing techniques [4] to obtain distance constrained topological mapping. This approach considers the network resources and the communication pattern generated by the tasks assigned to different cores in order to map such cores to the network nodes. The method is based on three main steps.

Step 1 Model the network as a table of distances (or costs) between each pair of source/destination nodes. The cost for communicating each pair of nodes is computed as inversely related to the available network bandwidth.

Step 2 Perform a heuristic search in the solution space with the aim of obtaining a near-optimal mapping that satisfies our distance restrictions.

Step 3 Repair the mapping found in the second step if some communicating pairs violate the distance constraints.

We have computed steps 1 and 2 as in our previous work ([4]). If the mapping found by the heuristic method does not satisfy the distance constraint, then a heuristic repair procedure, step 3, tries to repair the solution found. This procedure, based on [12], consists of a hill-climbing that minimizes the number of constraints violated by the solution mapping. The advantage of this procedure is that it is fast to compute, but presents some drawbacks like the capacity to fall in a local minimum that does not satisfy the distance constraint.

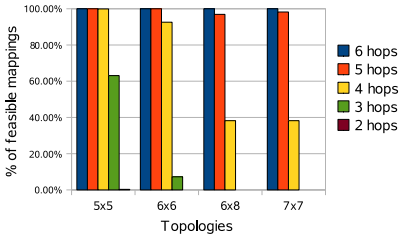


Fig. 2. Feasible mappings with distant constraint

It should be noticed that the goal of this work is to prove that it is possible to obtain efficient solutions for the problem of distance constrained mapping by using a mapping technique analogous to the one shown in our previous work ([4]). In order to achieve this goal, we have used the same heuristic method for solving step 2 than the one shown in [4]. Since that method was not designed for this problem, a new heuristic method specifically developed for solving this problem is likely to provide better solutions.

3.1 Feasibility Experiments

In order to test the distance constrained topological mapping method we have made a set of feasibility experiments. These consists of mapping 500 random APCGs on several 2-D mesh topologies. We have used a $5x5$, $6x6$, $6x8$ and $7x7$ mesh topologies. Each node of each APCG presents an out-degree log-normally distributed with a mean of 2 and a standard deviation of 1 communications. We have used an uniform probability distribution for spatial communications. It means that the probability of a core c_i communicating with a core c_j is the same for every core. The communication bandwith between each pair of communicating cores is distributed uniformly between 10 and 100 Kbytes/sec.

Figure 2 shows the result of the experiments. The X-axis shows the topologies and the Y-axis shows the percentage of feasible mappings. Each bar in the figure presents for each topology the percentage number of APCGs the method is able to map given a distance. As can be seen, 96% of the cases, the mapping technique is able to map the 500 APCGs with a distance of only 5 hops for all the topologies tried. It means that the path length of the header flit can be reduced from the diameter of the topology to 5 hops reducing the network overhead.

4 Efficient Path Computation for Source Routing

After the cores have been mapped on the NoC platform which supports distance constrained source routing, the next step will be to compute efficient paths for all the communicating pairs of cores. For each source core these paths will be stored as a table in the corresponding resource (core) to network interface (RNI). The RNI will use this table to append the path in the header flit of the packet.

Table 1. Best routing algorithm for a traffic type

Traffic Type	**Best Routing Algorithm**
Random Traffic	XY
Hot-Spot Traffic	Odd-Even
East-Dominated Traffic	West First
West-Dominated Traffic	East First
Transpose Traffic	Negative First

Beside avoiding deadlocks, the computed paths should also avoid congestion and uniformly distribute traffic among the links in the network as much as possible.

4.1 Routing Algorithm selection

A large number of deterministic and adaptive routing algorithms are available for deadlock free routing in mesh topology NoCs. The most famous among these are XY, Odd-Even, West First and North Last routing algorithms. XY is a deterministic routing algorithm and allows a single path between every pair of nodes. The other algorithms are partially adaptive routing algorithms and prohibit the packets to take certain turns, but allow path adaptivity for most pairs. It has been shown that no single routing algorithm provides best performance for all types of traffic. Table 1 gives relatively best routing algorithm for some specific types of communication traffic [6].

A traffic is called West-Dominated if majority of communication (considering number of communications and communication volume) is from east to west. We analyze and classify the traffic using the mapped APCG and select the most appropriate routing algorithm. The analysis uses the relative position of source and destination cores and the communication volume between pairs [6].

4.2 Path Computation

One can easily compute a path for each communication pair using the routing algorithm selected using the analysis in the previous subsection. In the case of deterministic routing the only path available is selected. In the case of partially adaptive routing algorithm the path is constructed by making a choice with a uniform probability at all intermediate routers where a choice among multiple admissible ports is available. Our study has shown that communication traffic type is rarely pure. For example, it is rare to have an application with pure West-Dominated traffic. To handle this we use adaptivity of the routing algorithm to balance load on the links and avoid/reduce congestion. Figure 3 describes a constructive algorithm to achieve this.

All the communications are sorted in ascending order according to their communication volume. Then in each iteration a path is computed for one communication using the selected routing algorithm. At every router where there exists a choice among multiple output ports, the port is selected if a lower load is already

```
Algoritm Path-Computations(RA, C, P)
  /* Implicit inputs: Mappings of cores to slots in topology */
  /* Inputs: RA - Routing Algorithm,
             C - Set of Communications {Ci, i = 1..N} */
  /* Outputs: Set of paths P = {Pi, i = 1..N, Pi used for Ci} */
  begin
    1. Initialize load on each link li = 0, i = 1..Number of links
    2. Order communications Ci, i = 1..N in ascending order based on
       communication volume
    3. For i = 1 to N do
         - Find a path Pi for Ci using RA considering
           current loads on various links and update the loads
       end for
  end
```

Fig. 3. Pseudocode of the algorithm for path computation

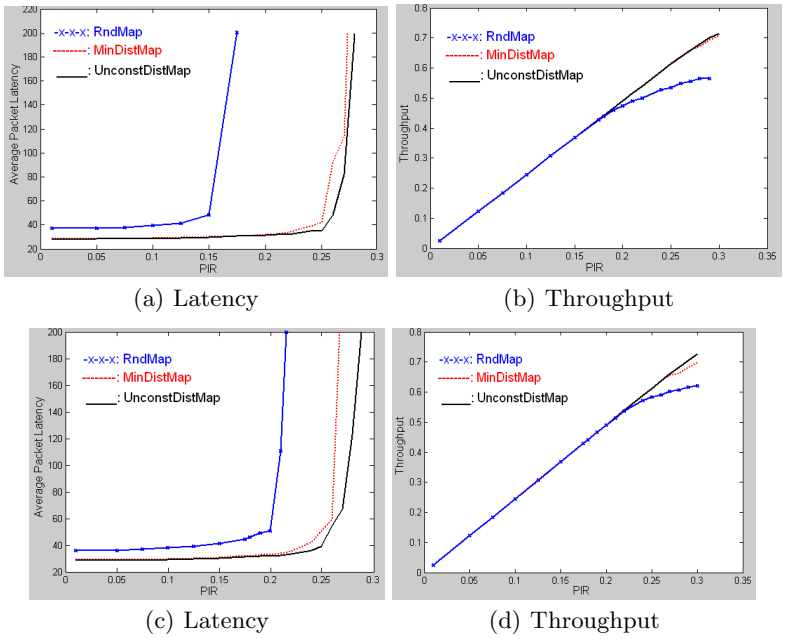
mapped to the corresponding output link. We keep updating the estimated load on links after each iteration. It has been shown that this methodology leads to efficient paths for communication [6].

5 Evaluation and Results

For evaluation purposes, we have evaluated the proposed approach using a set of random traffic scenarios. Each traffic scenario has been generated as described in section 3.1. For each scenario we have computed a random mapping, a near-optimal unconstrained mapping and a near-optimal constrained mapping for 5 hops. The Figure 4 shows the performance results in terms of latency and throughput for two of such traffic scenarios. In this figure the random mapping, the near-optimal unconstrained mapping and the near-optimal distance constrained mapping have been labeled as RNDMAP, UNCONSTDISTMAP and MINDISTMAP respectively.

The performance evaluation has been carried out using a NoC simulator developed in SDL language [6]. The simulator implements a NoC Platform based on a 2-D $7x7$ mesh topology and source routing. The simulated NoC also uses wormhole switching with a packet size fixed to 10 flits, the input buffers have capacity for keep 4 flits. We have used the source packet injection rate (pir) as load parameter. A Matlab based tool has been developed to compute efficient paths for source routing as described in Section 4. For each load value, latency and throughput values are averaged over 20,000 packets drained after a warm-up of 2000 packets drained.

Figure 4(a) and Figure 4(c) show the simulation results for average latency in cycles. These figures show that at both low traffic loads and high traffic loads both the UNCONSTDISTMAP and MINDISTMAP present similar behaviour and save more than 20% cycles and 25% over RNDMAP respectively.



(a) Latency (b) Throughput

(c) Latency (d) Throughput

Fig. 4. Simulation results for two random APCGs

The throughput results, measured in packets/cycle, are shown in Figure 4(b) and Figure 4(d). As we can look at the throughput achieved by the MINDISTMAP is almost the same as the throughput achieved by the UNCONSTDISTMAP and much higher than the throughput achieved by a RNDMAP close to saturation.

Therefore, the evaluation results show that is possible reduce the path length of the header flit at least to half of the network diameter without significantly degradation of the performance.

6 Conclusions and Future Work

We have addressed the application mapping problem for NoCs when the communication infrastructure is pre-designed as 2-D mesh topology using source routing and the path length header field is delimited up by a number of hops significantly less than the network diameter. In such a scenario we have demonstrated that our distance constrained mapping technique is able to map 96% of applications tried with a distance constraint less than half of the network diameter. We have proposed an efficient method, based on existing distributed deadlock free routing algorithms, to compute efficient paths required for source routing. The simulation based evaluation results show that our distance constrained mapping gives more than 20% latency improvement over random mapping at low traffic loads. The saturation points traffic load also shows around 25% . Performance degra-

dation as compared to path length constraint is negligible at low communication traffic and saturation packet injection rate is also only reduced by just 5%.

To the best of our knowledge this is the first attempt to consider core mapping for NoC platforms based on source routing. As future work, we plan to use intelligent heuristic search method to further lower the path length thus reducing bandwidth underutilization of NoC. We are also working on methods to improve computed paths for better link load balancing.

References

1. Benini, L., De Micheli, G.: Networks on chips: a new soc paradigm. Computer **35**(1) (Jan. 2002) 70–78
2. Chang, J.M., Pedram, M.: Codex-dp: co-design of communicating systems using dynamic programming. In: Proc. Design Automation and Test in Europe Conference and Exhibition 1999. (9–12 March 1999) 568–573
3. Murali, S., De Micheli, G.: Bandwidth-constrained mapping of cores onto noc architectures. In: Proc. Design, Automation and Test in Europe Conference and Exhibition. Volume 2. (16–20 Feb. 2004) 896–901
4. Tornero, R., Orduña, J.M., Palesi, M., Duato, J.: A communication-aware topological mapping technique for nocs. In: Euro-Par '08: Proceedings of the 14th international Euro-Par conference on Parallel Processing, Berlin, Heidelberg, Springer-Verlag (2008) 910–919
5. Mubeen, S., Kumar, S.: On source routing for mesh topology network on chip. In: SSoCC'09: 9th Swedish System on Chip Conference. (May 2009)
6. Mubeen, S.: Evaluation of source routing for mesh topology network on chip platforms. Master's thesis, School of Engineering, Jönköping University (June 2009)
7. Palesi, M., Holsmark, R., Kumar, S., Catania, V.: Application specific routing algorithms for networks on chip. IEEE Transactions on Parallel and Distributed Systems **20**(3) (2009) 316–330
8. Hu, J., Marculescu, R.: Energy-aware mapping for tile-based noc architectures under performance constraints. In: ASPDAC: Proceedings of the 2003 conference on Asia South Pacific design automation, New York, NY, USA, ACM (2003) 233–239
9. Goossens, K., Radulescu, A., Hansson, A.: A unified approach to constrained mapping and routing on network-on-chip architectures. Hardware/Software Codesign and System Synthesis, 2005. CODES+ISSS '05. Third IEEE/ACM/IFIP International Conference on (Sept. 2005) 75–80
10. Tornero, R., Starrantino, V., Palesi, M., Orduña, J.M.: A multi-objective strategy for concurrent mapping and routing in network on chip. In: Proceedings of the 23rd IEEE International Parallel and Distributed Processing Symposium. (25–29 May 2009)
11. Flich, J., López, P., Malumbres, M.P., Duato, J.: Improving the performance of regular networks with source routing. In: ICPP'00: Proceedings of the Proceedings of the 2000 International Conference on Parallel Processing, Washington, DC, USA, IEEE Computer Society (2000) 353
12. Minton, S., Johnston, M.D., Philips, A.B., Laird, P.: Solving large-scale constraint-satisfaction and scheduling problems using a heuristic repair method. In: AAAI. (1990) 17–24

Parallel Variable-Length Encoding on GPGPUs

Ana Balevic

IPVS, University of Stuttgart
ana.balevic@ipvs.uni-stuttgart.de

Abstract. Variable-Length Encoding (VLE) is a process of reducing the input data size by replacing fixed-length data words with codewords of shorter length. As VLE is one of the main building blocks in systems for multimedia compression, its efficient implementation is essential. The massively parallel architecture of modern general purpose graphics processing units (GPGPUs) has been successfully used for acceleration of inherently parallel compression blocks, such as image transforms and motion estimation. On the other hand, VLE is an inherently serial process due to the requirement of writing a variable number of bits for each codeword to the compressed data stream. The introduction of the atomic operations on the latest GPGPUs enables writing to the output memory locations by many threads in parallel. We present a novel data parallel algorithm for variable length encoding using atomic operations, which archives performance speedups of up to 35-50x using a CUDA1.3-based GPGPU.

1 Introduction

Variable-Length Encoding (VLE) is a general name for compression methods that take advantage of the fact that frequently occurring symbols can be represented by shorter codewords. A well known example of VLE, Huffman coding [1], constructs optimal prefix codewords on the basis of symbol probabilities, and then replaces the original symbols in the input data stream with the corresponding codewords.

The VLE algorithm is serial in nature due to data dependencies in computing the destination memory locations for the encoded data. Implementation of a variable length encoder on a parallel architecture is faced by the challenge of dealing with race conditions when writing the codewords to a compressed data stream. Since memory is accessed in fixed amounts of bits whereas codewords have arbitrary bit size, the boundaries between adjacent codewords do not coincide with the boundaries of adjacent memory locations. The race conditions would occur when adjacent codewords are written to the same memory location by different threads. This creates two major challenges for creating a parallel implementation of VLE: 1) computing destination locations for the encoded data elements with a bit-level precision in parallel and 2) managing concurrent writes of codewords to destination memory locations.

In recent years, GPUs evolved from simple graphics processing units to massively parallel architectures suitable for general purpose computation, also known

as GPGPUs. The nVidia GeForce GTX280 GPGPU used for this paper provides 240 processor cores and supports execution of more than 30,000 threads at once. In image and video processing, GPGPUs have been used predominantly for the acceleration of inherently data-parallel functions, such as image transforms and motion estimation algorithms [6–8]. The VLE entropy coding to our best knowledge has not been implemented on GPUs so far, due to its inherently serial nature. Some practical compression-oriented approaches on GPUs include compaction and texture compression. The compaction is a method for removing unwanted elements from the resulting data stream by using the parallel prefix sum primitive [9]. An efficient implementation of the stream reduction for traditional GPUs can be found in [10]. The texture compression is a fixed-ratio compression scheme which replaces several pixels by one value. Although it has a fast CUDA implementation [11], it is not suitable for codecs requiring a final lossless encoding pass, since it introduces a loss of fidelity.

We propose a fine-grain data parallel algorithm for lossless compression, and present its practical implementation on GPGPUs. The paper is organized as follows: Section II discusses related work, Section III presents an overview of the architecture of the GPGPU used for this study, in Section IV we present a design and implementation of a novel parallel algorithm for variable-length encoding (PAVLE), and in Section V, we present performance results and discuss effects of different optimizations.

2 GPGPU Architecture

The unified GPGPU architecture is based on a parallel array of programmable processors [5]. It is structured as a set of multiprocessors, where each multiprocessor is composed of a set of simple processing elements working in SIMD mode. In contrast to CPU architectures which rely on multilevel caches to overcome long system memory latency, GPGPUs use fine-grained multi-threading and a very large number of threads to hide the memory latency. While some threads might be waiting on data to be loaded from the memory, the fine-grain scheduling mechanism ensures that ready warps of threads (scheduling unit) are executed, thus providing effectively highly parallel computation resources.

The memory hierarchy of the GPGPU is composed of global memory (high-latency DRAM on the GPU board), shared-memory and register file (low-latency on-chip memory). The logical organization is as follows: the global memory can be accessed among all the threads running on the GPU without any restrictions; the shared memory is partitioned and each block of threads can be assigned one exclusive partition of the shared memory, and the registers are private to each thread. When GPU is used as a coprocessor, the data needs to be transferred first from the main memory of host PC to the global memory. In this paper, we will assume that the input data is located in the global memory, e.g. as a result of a computation or explicit data transfer from the PC.

The recent Tesla GPGPU architectures introduce hardware support for atomic operations. The atomic operations provide a simple method for safly handling

race conditions, which occur when several parallel threads try to access and modify data at the same memory location, since it is guaranteed that if an atomic instruction executed by a warp reads, modifies, and writes to the same location in global memory for more than one of the threads of the warp, each access to that memory location will occur and will be serialized, but the order in which they occur is not defined [13]. The CUDA 1.1+ GPU devices support the atomic operations on 32-bit and 64-bit words in the global memory, while CUDA 1.3 also introduces support for shared memory atomic operations.

3 The Parallel Variable-Length Encoding Algorithm

This section presents the parallel VLE (PAVLE) algorithm for GPGPUs with hardware support for atomic operations. The parallel variable-length encoding consists of the following parallel steps: (1) assignment of codewords to the source data, (2) calculation of the output bit positions for compressed data (codewords), and finally (3) writing (storing) codewords to the compressed data array. A high-level block-diagram of the PAVLE encoder is given in Fig. 1. Pseudocode for the

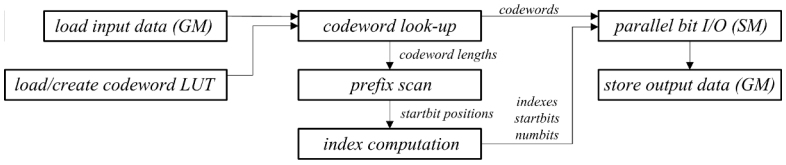


Fig. 1. Block diagram of PAVLE algorithm.

parallel VLE is given in Listing 1 with lines $2-5$ representing step 1, lines 68 being step 2 and lines $9-28$ representing step 3. The algorithm can be simplified if one assumes a maximal codeword length, as is done in the case for the JPEG coding standard. Restricting the codeword size reduces the number of control dependencies and also reduces the amount of temporary storage required, resulting in much greater kernel efficiency.

3.1 Codeword Assignment to Source Data

In the first step, variable-length codewords are assigned to the source data. The codewords can be either computed using an algorithm such as Huffman [4], or they can be predefined, e.g. as it is frequently the case in image compression implementations. Without loss of generality, we can assume that the codewords are available and stored in a table. This structure will be denoted as the codeword look-up table (codeword LUT). Each entry in the table contains two values: the binary code for the codeword, and codeword length in bits, denoted as a *(cw, cwlen)* pair. Our implementation uses an encoding alphabet of up to 256 symbols,

with each symbol representing one byte. During compression, each source data symbol (byte) is replaced with the corresponding variable-length codeword.

The PAVLE is designed in a highly parallel manner, with one thread processing one data element. The threads load source data elements and perform codeword look-up in parallel. As the current GPGPU architecture provides more efficient support for 32-bit data types, the source data is loaded as 32-bit unsigned integers. The 32-bit data values are split into four byte symbols, which are then assigned corresponding variable-length codewords from the codeword LUT. The codewords are locally concatenated into an aggregate codeword, and the total length of the codeword in bits is computed.

Algorithm 1 Parallel Variable Length Encoding Algorithm

```
 1: k ← tid
 2: for threads k = 1 to N in parallel
 3:     symbol ← data[k]
 4:     cw[k], cwlen[k] ← cwtable[symbol]
 5: end for
 6: for threads k = 1 to N in parallel
 7:     bitpos[1..N] ← prefixsum(cwlen[1..N])
 8: end for
 9: for threads k = 1 to N in parallel
10:     kc ← bitpos[k] div ws
11:     startbit ← bitpos[k] mod ws
12:     while cwlen[k] > 0 do
13:         numbits ← cwlen[k]
14:         cwpart ← cw[k]
15:         if startbit + cwlen > wordsize then
16:             overflow ← 1
17:             numbits ← wordsize − startbit
18:             cwpart ← first numbits of cw[k]
19:         end if
20:         put_bits_atomic(out, kc, startbit, numbits, cwpart)
21:         if overflow then
22:             kc ← kc + 1
23:             startbit ← (startbit + numbits) mod wordsize
24:             remove first numbits from cw[k]
25:             cwlen[k] ← cwlen[k] − numbits
26:         end if
27:     end while
28: end for
```

3.2 Computation of the Output Positions

To store the data which does not necessarily match the size of addressable memory locations, it is necessary to compute the destination address in the memory and also the starting bit position inside the memory location. Since in the

previous parallel step the codewords were assigned to input data symbols, the dependency in computation of the codeword output locations can be resolved based on the knowledge of the codeword lengths. The output parameters for each codeword are determined by computing the number of bits that should precede each codeword in the destination memory. The bit offset of each codeword is computed as a sum of assigned codeword lengths of all symbols that precede that element in the source data array. This can be done efficiently in parallel by using a prefix sum computation.

The prefix sum is defined in terms of a binary, associative operator $+$. The prefix sum computation takes as input a sequence $x_0, x_1, ..., x_{n-1}$ and produces an output sequence $y_0, y_1, ..., y_{n-1}$ such that $y_0 = 0$ and $y_k = x_0 + x_1 + ... + x_{k-1}$. We use a data-parallel prefix sum primitive [14] to compute the sequence of output bit offsets y_k on the basis of codeword lengths x_k, that were assigned to source data symbols. A work-efficient implementation of parallel prefix sum performs $O(n)$ operations in $O(\log n)$ parallel steps, and it is the asymptotically most significant component in the algorithmic complexity of the PAVLE algorithm. Given the bit positions at which each codeword should start in the com-

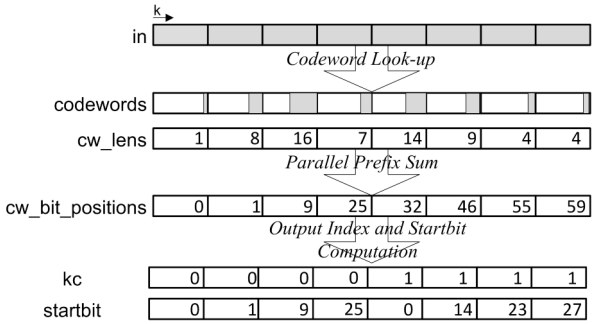


Fig. 2. An example of the variable-length encoding algorithm.

pressed data array in memory, the output parameters can be computed knowing the fixed machine word size, as given in the lines 10-11 of the pseudocode.It is assumed that the size of addressable memory locations is 32-bits, and it is denoted as *wordsize*. The variable k is used to denote the unique thread Id. It also corresponds to the index of data element processed by the thread k in the source data array. The *kc* denotes index of the destination memory word in compressed data array, and *startbit* corresponds to the starting bit position inside that destination memory word.

Fig. 2 is given as an illustration of the parallel computation of the output index and starting bit position on a block of 8 input data elements: The first two steps of the parallel encoding algorithm result in the generation of matching codewords for the input symbols, codeword lengths (as the number of bits), and output parameters for the memory writes to the output data stream. The

number of bits for each compressed data block is obtained as a byproduct of the first phase of the parallel prefix sum algorithm. Since a simple geometric decomposition is inherently applied on the GPUs as a step of the mapping process, this result can be used for concatenating the compressed data blocks into a contiguous array prior to data transfers from GPU to system memory.

3.3 Parallel Bit-Level Output

Bit-level I/O libraries designed for general-purpose CPUs process data serially, i.e., the codewords are stored one after the other into the memory. Implementation of a VLE on a parallel architecture introduces a problem of correctly dealing with race conditions that occur when different threads attempt to write their codewords to a same memory location. A recently introduced hardware support for atomic bitwise operations enables efficient execution of concurrent threads performing bit-level manipulations on the same memory locations, thus providing a mechanism for safely handling race conditions. The parallel output of codewords will produce correct results regardless of the write sequence, provided that each sequence of read-modify-write operations on a single memory location can be successfully completed without interruption, and that each output operation operation changes only the precomputed part of the destination word. The parallel bit-level output is executed in two stages: First, the contents

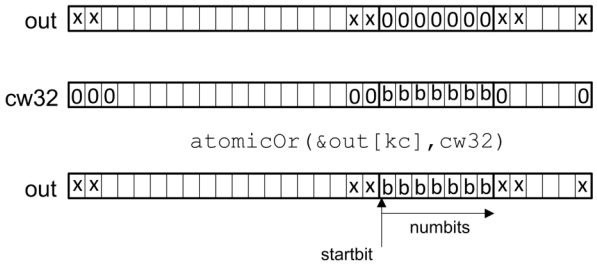


Fig. 3. Setting memory contents at index kc to the desired bit-values (codeword).

of the memory location at the destination address are prepared for the output by masking the *numbits* number of bits corresponding to the length of the codeword starting from the pre-computed bit output position. Second, the bits at these positions in the destination location are set to the value of the codeword, as illustrated in Fig. 3. If the contents of the destination memory are set in advance (all zeros), the output method can be reduced to only one atomic *or* operation. The implementation of the *put_bits_atomics* procedure for the GPGPUs supporting atomic operations (CUDA1.1+ compatible) is given in the code listing below.

A situation when a codeword crosses boundaries of a destination word in memory can occur during variable length encoding, e.g., when the *startbit* is near

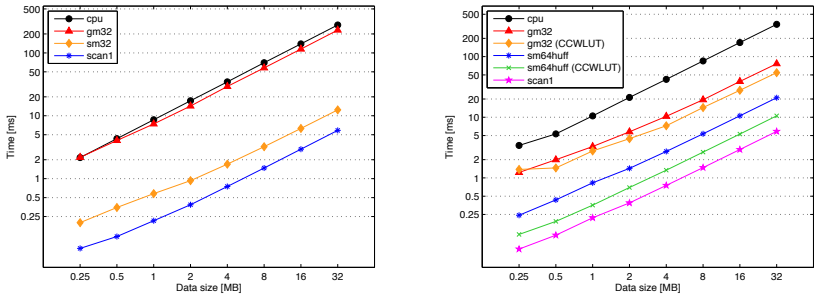
the end of the current word, and the codeword to be written requires more bits than what is available in the reminder of the current word. The crossing of the word-boundary is detected and handled by splitting the output of the codeword into two or more store operations. When the codeword cross boundaries of several machine words, some of the atomic operations can be replaced by the standard store operation. The inner parts of the codeword can be simply stored to the destination memory location(s), and only the remaining bits on both sides of the codeword need to be set using the atomic operations.

```
__device__ void put_bits_atomic(unsigned int* out, unsigned int kc,
                                unsigned int startbit, unsigned int numbits,
                                unsigned int codeword) {

    unsigned int cw32 = codeword;
    unsigned int restbits = 32-startbit-numbits;

#ifndef MEMSET0
    unsigned int mask = ((1<<numbits)-1);
    mask <<= restbits;
    atomicAnd(&out[kc], ~mask);
#endif

    if ((startbit == 0) && (restbits == 0)) out[kc] = cw32;
        else atomicOr(&out[kc], cw32 << restbits);
}
```

4 Performance Results

Performance of several kernel implementations was benchmarked on a PC with an 2.66 GHz Intel QuadCore CPU, 2 GB RAM memory, and a nVidia GeForce GTX280 GPU supporting atomic instructions on 32-bit words. The test data set was composed of randomly generated test data files of different sizes and different amount of information content (entropy between 0.5-8 bits/symbol). The test files were assigned variable-length codewords using the Huffman algorithm with the restriction on the maximal codeword length. The performance of a CPU encoder running on one 2.66GHz CPU core is given as a reference. Fig. 4(a) gives a performance comparison on a data set with 2.2 bits/symbol entropy. The GPU encoder *gm32* concatenates codewords for every 4 consecutive symbols (bytes) and writes the aggregate codeword to the GPU memory using global memory atomic operations. The performance of the serial encoder and the global memory (GM) encoder *gm32* are closely matching. However, by performing the atomic operations on a temporary buffer in shared memory (SM), as in *sm32*, a speed-up of more than an order of magnitude is achieved. The performance of the scan kernel, which is the asymptotically dominant part of the parallel algorithm, is given as a reference.

The *gm32* and *sm32* kernels operate under the assumption that the size of the aggregate codeword for four consecutive symbols (bytes) will not exceed



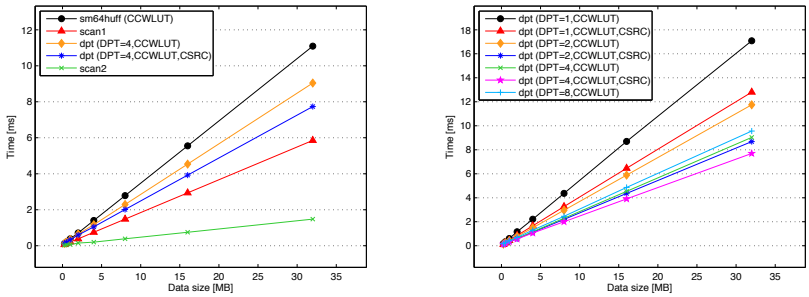
(a) CPU and GPU Kernels (b) Codeword LUT caching

Fig. 4. Kernel execution times as a function of data size.

the original data length, i.e. it will always fit into one 32-bit word. When using Huffman codewords, it may happen (although rarely) that the aggregate codeword exceeds the original data size. We designed a second SM kernel, denoted as *sm64huff*, that has a temporary buffer for the aggregate codeword of twice the original data size (a typical buffer size in compression implementations). The performance of *sm64huff* is slightly lower than the performance of *sm32* kernel, since it must perform one additional test during the codeword output. The situation when a codeword spans more than two destination memory locations is however correctly supported. In this case, no atomic operation is needed for the part of the codeword that spans an entire memory location, and a standard store operation can be used. However, empirical evaluation showed that atomic operations on the shared memory are implemented very efficiently, and that introduction of the additional test actually hurts the performance due to the increased warp serialization.

Additional performance improvements can be achieved by caching the codeword LUT, instead of looking up the codeword for each symbol in the global memory every time a symbol occurs. Fig. 4(b) gives a comparison of kernel execution times when the codeword look-ups are performed on the shared memory. Similar results are achieved by using the texture memory, which is cached by each multiprocessor. Use of low-latency shared memory for caching the codeword LUT improved the performance of GM kernels by approximately 20%, and the performance of SM kernels by up to 55%. As the symbols that appear more frequently are replaced by the codewords of shorter length, the low entropy data (well-compressible) will result in more shorter codewords that should be stored by different threads at the same memory location. This issue could be mitigated by processing more than one 32-bit data element per thread. The average number of bits that are written by each thread in one atomic operation to the destination memory location is increased and fewer atomic operations are issued.

Additionally, increasing DPT reduces the total number of data elements that is processed by the prefix sum (*scan*), which significantly influences the run time.



(a) Comparison of standard and DPT kernels

(b) DPT Parameter effects

Fig. 5. Effects of processing more data per thread (lin scale).

Fig. 5(a) shows performance gains using the ideal DPT value; performance of *scan* using the original and reduced number of blocks are given as a reference. Additional improvements are achieved by (1) caching the codeword LUT as previously described, and (2) caching aggregate codewords for every DPT elements in a local buffer. However, further increasing DPT radically increases memory requirements, since data is compressed in a shared memory buffer prior to transfer to the global memory. Fig. 5(b) gives a comparison of run times using several different DPT values. The investigation showed that the maximal DPT is limited by the shared memory requirements, and is relatively low ($DPT_{max} = 8$ when only codeword table is cached, and $DPT_{max} = 4$ when also aggregate codewords are cached). The best results are obtained using DPT= 4, resulting in a 35x speed-up.

5 Conclusion

In this paper, we presented a method for parallel bit-level output of data and a novel parallel algorithm for variable-length encoding (PAVLE) for GPGPU architectures supporting atomic operations. The PAVLE algorithm was implemented on a CUDA1.3 GPGPU using atomic operations on the shared memory for managing concurrent codeword writes, parallel prefix sum for computing the codewords offsets in compressed data stream and caching of the codeword look-up tables in the low-latency memory. The optimized version of PAVLE for CUDA 1.3 compatible GPGPUs achieves performance of approximately 4GB/sec using Huffman codes for encoding the data on the nVidia GeForce GTX280 GPGPU. We observed considerable speedups compared to the serial VLE on the state of the art PCs (up to 35x on 2.66GHz CPU, and up to 50x on a 2.40GHz CPU), thus making the PAVLE an attractive lossless compression algorithmic building block for GPU-based applications.

References

1. D. Huffman, "A method for the construction of Minimum-Redundancy codes," *Proceedings of the IRE*, vol. 40, no. 9, pp. 1098–1101, 1952.
2. S. W. Golomb, "Run length encodings," in *IEEE Transactions on Information Theory IT-12*, Juillet 1966, pp. 399–401.
3. J. Wen and J. Villasenor, "Reversible variable length codes for efficient and robust image and video coding," in *Proceedings Data Compression Conference*, 1998, pp. 471–480.
4. M. Atallah, S. Kosaraju, L. Larmore, G. Miller, and S. Teng, "Constructing trees in parallel," in *Proceedings of the first annual ACM symposium on Parallel algorithms and architectures*. ACM New York, NY, USA, 1989, pp. 421–431.
5. E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym, "NVIDIA Tesla: A unified graphics and computing architecture," *Micro, IEEE*, vol. 28, no. 2, pp. 39–55, 2008.
6. Y. Allusse, P. Horain, A. Agarwal, and C. Saipriyadarshan, "GpuCV: an opensource GPU-accelerated framework forimage processing and computer vision," 2008.
7. W. Chen and H. Hang, "H. 264/AVC motion estimation implmentation on Compute Unified Device Architecture (CUDA)," in *2008 IEEE International Conference on Multimedia and Expo*, 2008, pp. 697–700.
8. J. Fung and S. Mann, "Using graphics devices in reverse: GPU-based image processing and computer vision," in *2008 IEEE International Conference on Multimedia and Expo*, 2008, pp. 9–12.
9. G. E. Blelloch, "Prefix sums and their applications," *Synthesis of Parallel Algorithms*, pp. 35—60, 1990.
10. D. Roger, U. Assarsson, and N. Holzschuch, "Efficient stream reduction on the gpu," in *Workshop on General Purpose Processing on Graphics Processing Units*, D. Kaeli and M. Leeser, Eds., Oct 2007.
11. Ignacio Castaño, "High quality dxt compression using cuda," last access: May, 2008.
12. A. Obukhov, "Discrete cosine transform for 8x8 blocks with cuda," last access: May, 2009.
13. NVIDIA Corporation Technical Staff, "Nvidia cuda -programming guide 2.0,"last access: Dec, 2008. [Online]. Available: `http://developer.download.nvidia.com/compute/cuda/`
14. M. Harris, S. Sengupta, and J. D. Owens. Parallel prefix sum (scan) with cuda. *GPU Gems*, March, 2007.

Towards Metaprogramming for Parallel Systems on a Chip

Lee Howes[1], Anton Lokhmotov[1], Alastair F. Donaldson[2], and Paul H. J. Kelly[1]

[1] Department of Computing, Imperial College London,
180 Queen's Gate, London, SW7 2AZ, UK
[2] Computing Laboratory, University of Oxford,
Parks Road, Oxford, OX1 3QD, UK

Abstract. By presenting implementations of several versions of an image processing filter, evaluated on Intel and AMD multicore systems equipped with NVIDIA graphics cards, we demonstrate that efficiently implementing an algorithm to execute on commodity parallel hardware requires careful tuning to match the hardware characteristics. While such manual tuning is possible, it is not practical: the number of versions to write and maintain grows with the number of target architectures, becoming infeasible for large applications. Our findings motivate the need for tools and techniques that decouple a high level algorithm description from low level mapping and tuning. We believe that the issues that make such mapping and tuning difficult can be reduced by allowing the programmer to describe both execution constraints and memory access patterns using *Æcute metadata*, a high level representation which we briefly describe.

1 Introduction

We describe implementations of several versions of a simple image processing filter which we evaluate on x86 multicore systems and a GPU-accelerated system via thorough design space exploration. Our experimental results demonstrate that efficiently implementing an algorithm to execute on commodity parallel hardware requires careful tuning to match the hardware characteristics, as even similar systems show a variation in performance depending on low-level details such as iteration space tiling and data layout. While such manual tuning is possible, it is not practical: the number of versions to write and maintain grows with the number of target architectures. For applications consisting of multiple kernels such development and maintenance becomes infeasible.

Our findings motivate the need for tools and techniques that decouple a high level algorithm description from low level mapping and tuning. We believe that the issues that make such mapping and tuning difficult can be reduced by allowing the programmer to describe both execution constraints and memory access patterns using a high level representation such as *Æcute metadata* [1], an example of which we briefly present.

2 Vertical mean image filter

We consider a vertical mean image filter, for which the output pixel at position (x, y) is given by the formula

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

- $\mathbf{I}$ is a $W \times H$ grey-scale input image;
- $\mathbf{O}$ is a $W \times (H - D)$ grey-scale output image;
- D is the *diameter* of the filter, *i.e.* the number of input pixels over which the mean is computed (typically, $D \ll H$);
- $0 \leq x < W, 0 \leq y < H - D$.

Mean filtering is a simple technique for smoothing images, *e.g.* to reduce noise.

Let N be the number of output pixels: $N = W \times (H - D)$. A naïve parallel algorithm can run N threads, each producing a single output pixel, which requires $\Theta(ND)$ reads and arithmetic operations. A good parallel algorithm, however, must be efficient and scalable [2].

2.1 Scalable algorithm

The algorithm in Listing 1 *strips* the computation in the vertical dimension, where up to T outputs in the same strip are computed serially in two *phases*. The first phase in lines 6–10 computes $\mathbf{O}_{x,y0}$ according to (1). The second phase in lines 12–19 computes $\mathbf{O}_{x,y}$ for $y \geq y0 + 1$ as $\mathbf{O}_{x,y-1} + \big(\mathbf{I}_{x,y+D-1} - \mathbf{I}_{x,y-1}\big)/D$.

```
// for each column                                                    1
for(int x = 0; x < W; ++x)                                            2
{ // for each strip of rows                                          3
  for(int y0 = 0; y0 < H-D; y0 += T)                                  4
  {                                                                   5
    // first phase: convolution                                       6
    float sum = 0.0f;                                                 7
    for(int k = 0; k < D; ++k)                                        8
      sum += I[(y0+k)*W + x];                                         9
    O[y0*W + x] = sum / (float)D;                                     10
                                                                      11
    // second phase: rolling sum                                      12
    for(int dy = 1; dy < min(T,H-D-y0); ++dy)                         13
    {                                                                 14
      int y = y0 + dy;                                                15
      sum -= I[(y-1)*W + x];                                          16
      sum += I[(y-1+D)*W + x];                                        17
      O[y*W + x] = sum / (float)D;                                    18
    }                                                                 19
  }                                                                   20
}                                                                     21
```

Listing 1: Vertical mean image filter algorithm in C.

This algorithm performs $\Theta(N + ND/T)$ reads and arithmetic operations, significantly reducing memory bandwidth and compute requirements for $T \gg D$. Since the x and $y0$ loops carry no dependences, up to $\lceil N/T \rceil$ threads can run in parallel.

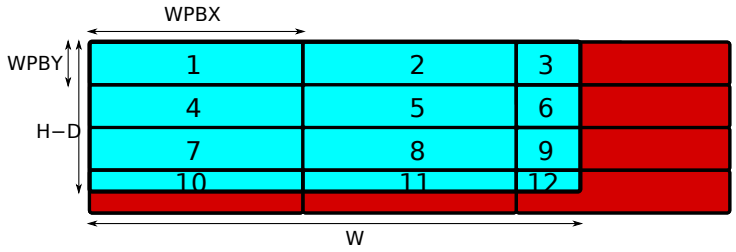
Note that since the order of arithmetic operations is undefined in (1), both the naïve and scalable algorithms are functionally, if not arithmetically, equivalent.

Clearly, the optimal value of T depends on problem parameters (W, H and D), and device parameters (*e.g.* the number of cores and memory partitions). Thus, in §2.3, we use the approach of *iterative compilation* to find the optimum.

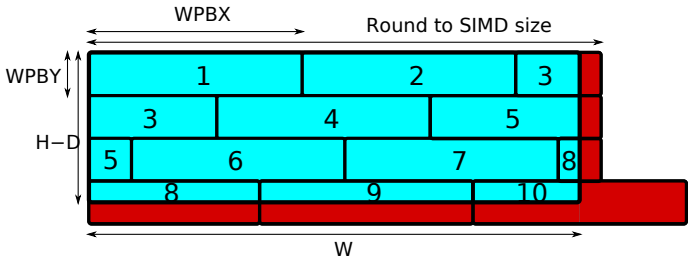
2.2 Implementation

We describe efficient implementations of the vertical mean filter for a GPU, using the NVIDIA Compute Unified Device Architecture (CUDA) [3], and for a multicore CPU using Intel Streaming SIMD Extensions (SSE) [4].

CUDA Implementing the vertical mean filter efficiently on a GPU requires mapping the iteration space onto threads, which are grouped into blocks located in a grid.



(a) A 2D grid mapping loses efficiency from unused threads off the right image edge.



(b) A 1D grid mapping uses its threads more efficiently by wrapping around the right image edge. For efficiency, it must take into account alignment, which complicates both memory access and iteration.

Fig. 1: Different mapping strategies result in different utilisation of threads. Light and dark regions of blocks denote used and unused threads, respectively.

The most natural iteration space mapping is into thread blocks on a 2D grid, with each block producing a rectangular section of the output image, of size WPBX $\times$ WPBY (WPB stands for *work per block*). However, if the image width is not a multiple of WPBX, significant portions of thread blocks covering the right edge of the image may be unused, as illustrated by Figure 1a.

This issue can be alleviated by mapping into thread blocks on a 1D grid that covers the image by wrapping around the right edge, as illustrated by Figure 1b. As we show in §2.3, a mapping that maximises thread utilisation suffers from misalignment, if the image width is not a multiple of the size of the SIMD unit (*warp* in NVIDIA's terminology); a better mapping takes alignment into account by wasting a small number of threads on the right of the image, thus ensuring that the first pixel of each row is handled by the first thread in a SIMD unit.

SSE On a multicore CPU, the notion of a GPU thread block corresponds to an instruction stream running on the vector unit of a single core, and a thread to an individual vector lane. Since the number of threads is relatively low, one approach is to assign an entire output column of pixels to a single thread, so that $T = H - D$. For a machine with C cores with SIMD width S, there is no advantage to using more than $C \times S$ threads because load balancing is not a concern for the algorithm and a high degree of parallelism to cover memory latency is unnecessary. The out of order execution logic of the CPU obtains adequate parallelism at the instruction level from a set of C threads (a single CPU thread).

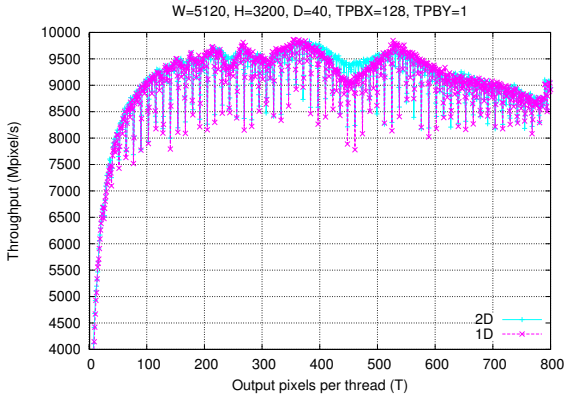
When vectorising code in Listing 1 on a single core, the loop x must be interchanged with the loop $y0$ (which will only execute once if $T = H - D$), and stripmined into vectors (of 4 elements in SSE [4]). The vectorised code can then be parallelised across multiple cores, vertically ($T < H - D$) or horizontally ($T = H - D$), in a straightforward manner such that a single instruction stream processes pixels from contiguous columns (to gain from cache and prefetching mechanisms within each core).

2.3 Experimental results

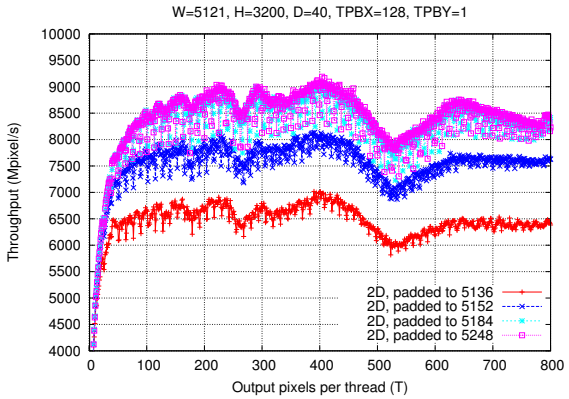
CUDA Figure 2 presents experimental results obtained on a dual-core 3GHz Intel Core 2 Duo E8400 system with 2GiB RAM, equipped with an NVIDIA GTX 280 card, running 64-bit Linux Ubuntu 8.04. Code is compiled using CUDA SDK 2.2 and GCC 4.2.4 with the "-O3" optimisation setting. We measure the kernel execution time only and record the best throughput out of 50 runs. Parameter TPBX/TPBY records the number of threads per block in the X/Y dimension.

In all the experiments, we fix the number of threads per block at 128 (128×1), as we nearly achieve the peak memory efficiency with this setting: $\approx$ 10 Gpixel/s $\times$ 4 bytes/pixel $\times$ (2 reads + 1 write) = 120 GB/s (close to the bandwidth of aligned copy on this card). Thus, WPBX = 128 and WPBY = T.

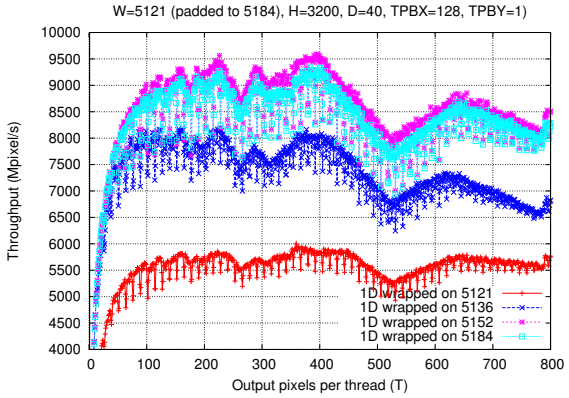
Figure 2a shows that the 1D and 2D grid versions are similar in throughput when applied to a 5120×3200 image, where 5120 is a multiple of 128 pixels. The throughput is below 800 Mpixel/s when each thread produces a single pixel, climbs fast with increasing serial efficiency, achieving (by the 1D grid version) the peak throughput of 9.89 Gpixel/s when $T = 355$, and then declines with decreasing parallelism.



(a) 5120×3200 image. 2D grid; 1D grid.



(b) 5121×3200 image. 2D grid. Data padded to multiples of 16, 32, 64, and 128 pixels.



(c) 5121×3200 image. Data padded to 5184 (a multiple of 64) pixels. 1D grid wrapped on the image width and multiples of 16, 32 and 64 pixels.

Fig. 2: Comparison of different mappings with various image sizes, data padding and thread wrapping alignment.

When applied to a 5121×3200 image, however, the 2D grid version only achieves 7.02 Gpixel/s, as shown by the bottom line in Figure 2b. Whilst we allocate memory using the `cudaMallocPitch` function, which pads the image to a multiple of 16 pixels to enable global memory access coalescing (5136 pixels in this case), such allocation leads to DRAM partition conflicts. We remedy the conflicts by manually padding the image to a multiple of 32, 64 and 128. Since the results of padding to a multiple of 64 and 128 are barely distinguishable, we fix the image padding at a multiple of 64 (5184 pixels) for all subsequent experiments.

Figure 2c shows that the 1D grid mapping that maximises thread utilisation by wrapping on 5121 pixels only achieves 6.00 Gpixel/s, whilst wrapping on the image padding of 5184 pixels performs worse than wrapping on the warp size multiple of 5152 pixels.

To summarise, for the misaligned image padded to 5184 pixels, the 1D grid version wrapped on 5152 pixels achieves 9.58 Gpixel/s at $T = 396$, whilst the 2D grid version achieves only 9.06 Gpixel/s at $T = 409$; thus, the 1D grid version performs 6% better than the 2D grid one.

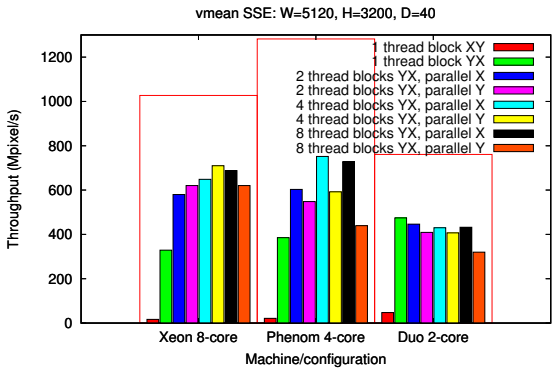


Fig. 3: Comparison of different blocking strategies for the CPU version of the vertical mean filter. The large surrounding boxes represent the peak memory copy throughput for each of the architectures, as obtained by running the STREAM benchmark [5].

SSE Figure 3 presents experimental results obtained on: 2.5GHz eight-core (dual-socket quad-core) Intel Xeon E5420 with 16GiB RAM (Xeon); 2.3GHz quad-core AMD Phenom 9650 with 8GiB RAM (Phenom); 3GHz dual-core Intel Core 2 Duo E8400 with 2GiB RAM (Duo). All systems run 64-bit Linux Ubuntu 8.04. Code is compiled using Intel Compiler 11.0 with the "-xHost -fast" optimisation settings.

As a baseline for comparison we use a version where the horizontal and vertical loops have not been interchanged to lead to contiguous memory accesses. We refer to this version as XY, and to versions where loop interchange has been applied as YX.

The YX loop scans horizontally and sums into an intermediate accumulation array. We compare the number of thread blocks (cpu threads) and the way the computation is parallelised – horizontally ("parallel X") or vertically ("parallel Y"). Using more thread blocks than cores performs worse as load balancing issues are minimal and overhead is increased. The peak throughput is sometimes achieved with a lower number of thread blocks than the number of cores (*e.g.* with 4 threads on the eight-core Xeon system). Another peculiarity is that parallelising horizontally ("parallel X") is always more beneficial than parallelising vertically ("parallel Y") on the AMD Phenom system, whilst this is not the case on the Intel systems.

3 Towards metaprogramming

To ease the programmer's burden of mapping and tuning computation kernels to parallel systems on a chip, we propose extending a kernel's description with decoupled Access/Execute (Æcute) metadata. Execute metadata for a kernel describes its iteration space ordering and partitioning. Access metadata for a kernel describes memory locations the kernel may access on each iteration.

This access and execute metadata improves productivity and portability of programming for the following reasons. First, the specification of the iteration space ordering and partitioning is independent of the addressing of data or the computation kernel. Addressing and iteration space visiting (loop) code can be generated automatically. Partitioning of a program can be device specific, either programmer specified or discovered via design space search. Secondly, data movement code can be generated from memory access pattern descriptions [1]. Efficient data movement code to deal with alignment and synchronisation constraints can be complicated and time-consuming to produce by hand. In particular, on vector architectures, independent computation elements (CUDA threads) cooperate in one way for movement, and another for computation. This breaks the thread separation model for CUDA, and cuts across other optimisations inconveniently.

```
// Array descriptors (C array wrappers)                          1
Array2D<float> arrayI(&I[0][0], W, H);                           2
Array2D<float> arrayO(&O[0][0], W, H-D);                         3
                                                                 4
// Execute metadata: parallel iteration space                    5
IterationSpace1D x(0,W);                                         6
IterationSpace1D y(0,H-D);                                       7
IterationSpace2D iterXY(x,y);                                    8
                                                                 9
// Access metadata: iteration space -> memory                    10
VerticalStrip2D_R accessI(iterXY, arrayI, D);                    11
Point2D_W accessO(iterXY, arrayO);                               12
```

Listing 2: Æcute metadata for the vertical mean image filter.

We give an example of Æcute metadata for the vertical mean image kernel in Listing 2. In lines 1–3 we wrap accesses to plain C arrays `I[W][H]` and `O[W][H-D]` into

Æcute array descriptors `arrayI` and `arrayO` to cleanse the kernel of uncontrolled side-effects. In lines 5–8 we construct a 2D iteration space descriptor `iterXY` from 1D descriptors `x` and `y`, having the same bounds as the output image dimensions. By default, an iteration space is parallel in every dimension. Finally, in lines 10–12 we specify that on each iteration of the 2D iteration space the kernel reads a vertical strip of D pixels from `arrayI` and writes a single pixel to `arrayO`.

Similar to Stanford's Sequoia language [6], we target systems with software-managed memory hierarchies and seek to separate a high-level algorithm representation from a system-specific mapping. Unlike Sequoia, we base our mapping on partitioning (manually or automatically) an iteration space into disjoint subspaces and infer memory access of subspaces from Æcute metadata.

For example, for a single GPU-accelerated system, a hierarchy of iteration space partitions can specify subspaces to be executed:

- at the lowest level, by individual threads:
```
// 1xT outputs per thread
iterXY.partitionThreads(1,T);
```
- at the middle level, by blocks of possibly cooperating threads:
```
// 128xT outputs per block
iterXY.partitionBlocks(128,T);
```
- at the highest level, by possibly cooperating compute devices:
```
// (W/2)x(H-D) outputs per device
iterXY.partitionDevices(W/2,H-D)
```

Further extensions are possible onto clusters of accelerated nodes.

4 Work in progress

The OpenCL initiative [7] aims to provide portability across heterogeneous compute devices by providing a detailed, low-level API for describing computational kernels. Although a standard-compliant OpenCL kernel will execute correctly on any standard-compliant implementation, it is clear that performance of the kernel will depend critically on characteristics of the underlying hardware.

We are working on a tool that will take a high-level algorithm representation and generate efficient device-specific OpenCL code. The representation will be kept similar to C++, *e.g.* as in Listing 1 with accesses to C arrays replaced with accesses to Æcute array descriptors as in Listing 2. Code generation will be particularly oriented towards effectively orchestrating data movement in software-managed memory hierarchies, including automatically handling such low-level details as data alignment and padding.

5 Related work

In previous work we introduced the Æcute framework for explicit but separate expression of the memory access pattern and execution schedule for a computational kernel, showing that Æcute specifications can be used for automatic generation of data-movement code which performs within a reasonable margin of hand-tuned code on the

Cell BE processor [1]. The hand-written SSE and CUDA versions of the vertical mean filter which we consider in this paper are examples of possible outputs from an extended Æcute framework. We plan to extend the ideas of [1] with the lessons learned from this case-study, leading to a generative approach to programming highly parallel systems.

We have explored the optimization space of the vertical mean filter example by running large batches of experiments for different image sizes and partitions. A more sophisticated method of optimization space exploration, termed optimization space *carving* is presented in [8].

The CUDA-lite [9] experimental enhancement to CUDA aims to simplify GPU programming, allowing the programmer to write in a more abstract version of CUDA which hides the complex memory hierarchy of a GPU, and providing a source-to-source translation resulting in equivalent CUDA code which aims to conserve memory bandwidth and reduce memory latency.

SPIRAL [10] aims to generate efficient DSP transformations on various architectures, including GPUs, from high level mathematical representations. It uses representations forms of formulae that allow efficient code generation for specific architectures and attempts to find a mathematical transformation, through heuristics and auto-tuning, to utilise these formulae. SPIRAL's representations are more mathematical and domain-specific than those used in the Æcute model but offer hints about how Æcute kernels could be combined in sequences with iteration space reordering.

6 Conclusions and future work

By exploring the optimization space of several versions of an image processing filter, evaluated on Intel and AMD architectures equipped with GPUs, we have shown that the strategy used for iteration space partitioning can have a dramatic effect on the performance of the filter. No single version is suitable for both GPU and CPU architectures, and each version requires quite different code to be written and maintained. We have considered only a simple image processing kernel; clearly the difficulty of creating and maintaining multiple versions of efficient code bases only increases in difficulty when working with more complicated kernels and full HPC applications.

The cost and maintenance problem makes it problematic for programmers to work at this level of development. We plan to investigate extensions to the Æcute model [1] to support automatic code generation and optimisation for GPUs, CPUs with vector capabilities, and multi-core processors with scratch-pad memories such as the Cell Broadband Engine. We believe that cleanly separating the execution schedule of a kernel from its memory access pattern has the potential to facilitate productive and efficient programming of heterogeneous multi-core systems.

References

1. Howes, L.W., Lokhmotov, A., Donaldson, A.F., Kelly, P.H.: Deriving efficient data movement from decoupled Access/Execute specifications. In: Proceedings of the 4th International conference on High-Performance Embedded Architectures and Compilers (HiPEAC). Volume 5409 of LNCS., Springer (2009) 168–182

2. Lin, C., Snyder, L.: Principles of Parallel Programming. 1st edn. Addison-Wesley, Boston, MA, USA (2008)
3. NVIDIA: CUDA. `http://www.nvidia.com/cuda` (2006–2009)
4. Bik, A.J.: The Software Vectorization Handbook. Applying Multimedia Extensions for Maximum Performance. Intel Press (2004)
5. McCalpin, J.D.: STREAM: Sustainable memory bandwidth in high performance computers. `http://www.cs.virginia.edu/stream/` (1990–2009)
6. Fatahalian, K., Horn, D.R., Knight, T.J., Leem, L., Houston, M., Park, J.Y., Erez, M., Ren, M., Aiken, A., Dally, W.J., Hanrahan, P.: Sequoia: programming the memory hierarchy. In: Proceedings of the ACM/IEEE conference on Supercomputing. (2006) 83
7. The Khronos Group: OpenCL. `http://www.khronos.org/opencl` (2008–2009)
8. Ryoo, S., Rodrigues, C.I., Stone, S.S., Stratton, J.A., Ueng, S.Z., Baghsorkhi, S.S., Hwu, W.m.W.: Program optimization carving for GPU computing. J. Parallel Distrib. Comput. **68**(10) (2008) 1389–1401
9. Ueng, S.Z., Lathara, M., Baghsorkhi, S.S., Hwu, W.M.W.: CUDA-Lite: Reducing GPU programming complexity. In: LCPC'08. Volume 5335 of LNCS., Springer (2008) 1–15
10. Püschel, M., Moura, J.M.F., Johnson, J., Padua, D., Veloso, M., Singer, B., Xiong, J., Franchetti, F., Gacic, A., Voronenko, Y., Chen, K., Johnson, R.W., Rizzolo, N.: SPIRAL: Code generation for DSP transforms. Proceedings of the IEEE, special issue on "Program Generation, Optimization, and Adaptation" **93**(2) (2005) 232– 275

Dynamic detection of uniform and affine vectors in GPGPU computations

Sylvain Collange[1], David Defour[1] and Yao Zhang[2]

[1] ELIAUS, Université de Perpignan, 66860 Perpignan, France
{sylvain.collange,david.defour}@univ-perp.fr
[2] ECE Department, University of California Davis
yaozhang@ucdavis.edu

Abstract. We present a hardware mechanism which dynamically detects uniform and affine vectors used in SPMD architecture such as Graphics Processing Units, to minimize pressure on the register file and reduce power consumption with minimal architectural modifications. A preliminary experimental analysis conducted with the Barra simulator shows that this optimization can benefit up to 34 % of register file reads and 22 % of the computations in common GPGPU applications.

1 Introduction

GPUs are now powerful and programmable processors that have been used to accelerate general-purpose tasks other than graphics applications. These processors rely on a Single Program Multiple Data (SPMD) programming model. This model is implemented by many vector units working in a Single-Instruction Multiple-Data (SIMD) fashion, and vector register files. Register usage is a critical issue as the number of instance of the same program that can be executed simultaneously depend on the number of hardware registers and the register usage per instance. Making this bad situation worse, vectorizing scalar operations in an application makes inefficient use of registers and functional units. To efficiently handle scalar data, Cray-like vector machines incorporate scalar functional units as well as scalar registers. Modern GPUs lack such scalar support, leaving it to vector units. These vector units execute the same instruction on the same data leading to as many unnecessary operations as the length of the vector when uniform data are encountered. These unnecessary operations involve data transfers and activity in functional units that consume power, which is a critical concern in architectural and microarchitectural designs of GPUs.

We observed through our experiments that standard GPGPU applications use a significant number of vectors to store uniform data. A closer look at the manipulated values shows that this number is even higher when we consider affine values (e.g. 1, 3, 5, 7...) stored in a given vector. Motivated by this observation, we propose and evaluate by simulation a technique that tags a vector register file according to the type of registers: uniform, affine or generic vector.

The rest of the paper begins with a brief description of the NVIDIA architecture upon which our model is based. Section 3 presents our performance

evaluation methodology, based on a functional simulator named Barra. We use it to both evidence the presence of redundancy in calculations in Section 4 and evaluate the proposed technique described in Section 5. We discuss technical issues in Section 6. Section 7 presents quantitative results and figures, and Section 8 concludes the paper.

2 Architecture Model

The base architecture we consider in our simulations consists of a vector processor, a set of vector register files, a set of vector units, and an instruction set architecture that mimics the behavior of the NVIDIA GPUs used in the Compute Unified Device Architecture (CUDA) environment [1]. This environment relies on a stack composed of an architecture, a language, a compiler, a driver and various tools and libraries.

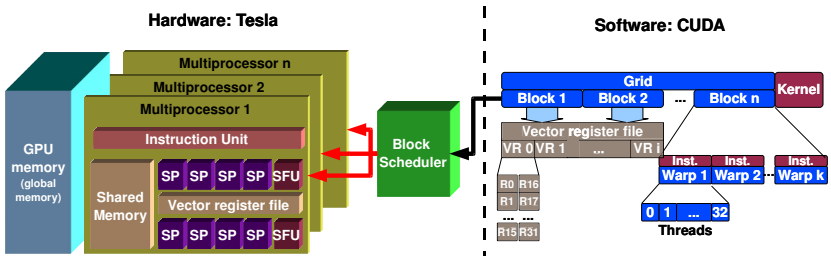


Fig. 1. Processing flow of a CUDA program.

A CUDA program runs on an architecture composed of a host processor CPU, a host memory and a graphics card with an NVIDIA GPU with CUDA support. All current CUDA-enabled GPUs are based on the Tesla architecture, which is made of an array of *multiprocessors*. Tesla GPUs execute thousands of threads in parallel thanks to the combined use of multiple multiprocessors, SIMD processing and hardware multithreading [2]. Figure 1 describes the hardware organization of such a processor. Each multiprocessor contains the logic to fetch, decode and execute SIMD instructions which operate on vectors of 32 elements. There are 256 or 512 vector registers, each register being a 32-wide vector of 32-bit values. In addition to the register file, each multiprocessor contains a scratchpad memory (or shared memory, using NVIDIA's terminology) and separate caches for constant data and instructions.

The hardware organization is tightly coupled with the parallel programming model of CUDA. The programming language used in CUDA is based on C with extensions to indicate if a function is executed on the CPU or the GPU. Functions executed on the GPU are called *kernels*. CUDA lets the programmer define if a variable resides in the GPU address space and specify the kernel execution across different granularities of parallelism: *grids*, *blocks* and *threads*. As the underlying hardware is a SIMD processor, threads are grouped together in so-called *warps* which operate on 32-wide vector registers. Each instruction is executed on a

warp by a multiprocessor. Warps execute instructions at their own pace, and multiple warps can run concurrently on a multiprocessor to hide latencies of memory and arithmetic instructions. This technique helps hide the latency of streaming transfers, and improve the effective memory bandwidth. The register file of a multiprocessor is logically split between the warps it executes. As a GPU includes several multiprocessors, warps are grouped into blocks. Blocks are scheduled on the available multiprocessors. A multiprocessor can process several blocks simultaneously if enough hardware resources (registers and shared memory) are available.

The compilation flow of a normal CUDA program is a three-step process directed by the CUDA compiler *nvcc*. First, according to specific CUDA directives from the CUDA Runtime API, the program is split into a host program and a device program. The host program is then compiled using a host C or C++ compiler and the device program is compiled through a specific back-end for the GPU. The resulting device code is binary instruction code (in *cubin* format) to be executed on a specific GPU. The host program and the device program are linked together using the CUDA libraries, which includes the necessary functions to load a cubin either from inside the executable or from a stand-alone file and send it to the GPU for execution.

3 Barra, a Functional Simulator of NVIDIA GPUs

Several options exist to model the dynamic behavior of CUDA programs. CUDA offers a built-in emulation mode that run POSIX threads on the CPU on behalf of GPU threads, thanks to a specific compiler back-end. However, this mode differs in many ways with the execution on a GPU: the behavior of floating-point and integer computations, the scheduling policies and memory organization are different.

GPU simulators running CUDA's intermediate language PTX such as GPGPU-Sim [3] or Ocelot [4] can offer a greater accuracy, but still run an unoptimized intermediate code instead of the instructions actually executed by a GPU.

Recent versions of CUDA include a debugger that allows watching the values of GPU registers between each line of source code. Though this mode offers perfect functional accuracy, it cannot be modified for instrumentation or feature evaluation purposes.

Barra [5] simulates the actual instruction set of the NVIDIA Tesla architecture at the functional level. The behavior of all instructions is reproduced with bit-accuracy, with the exception of transcendentals (exp, log, sin, cos, rcp, rsq). To our knowledge, Barra is the only publicly-available tool that both executes the same instructions as Tesla GPUs and allows viewing the exact contents of registers during the execution.

This simulator consists of two parts: a driver, and a simulator. The driver is a shared library with the same name and exporting the same symbols as NVIDIA's *libcuda.so* so that CUDA Driver API calls can be dynamically redirected to the simulator. It includes major API functions to load, and execute a CUDA program

and manage data transfers. The simulator takes the binary code compiled by NVIDIA's nvcc compiler as input simulates the execution of the kernel, and produces statistics for the each instruction.

3.1 Logical execution pipeline

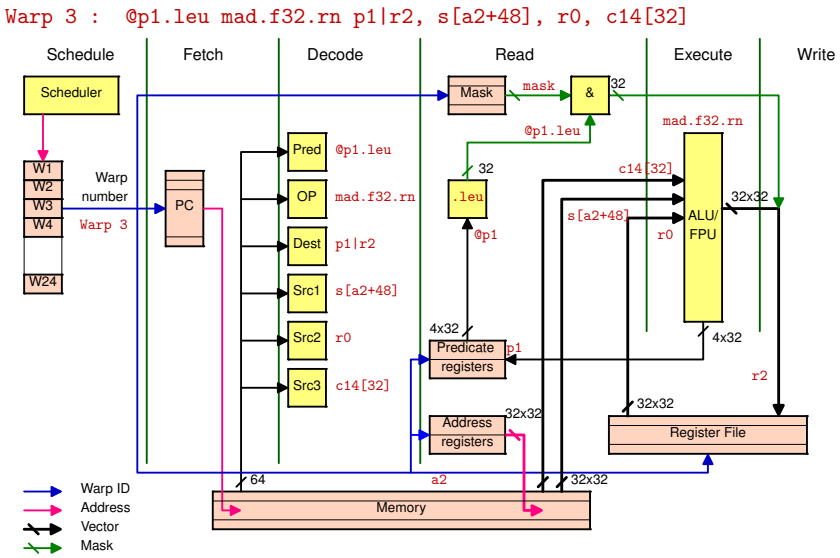


Fig. 2. Overview of the functional execution pipeline during the execution of a MAD instruction.

The instruction-set simulator executes each assembly instruction according to the model described in Figure 2. First, a scheduler selects the warp ready for execution according to a round-robin policy and reads its current program counter (PC). Then the instruction is fetched and decoded. Then operands are read from the register file or from on-chip memories (scratchpad) or caches (constants). The instruction is executed and its results are written back to the register file.

3.2 Vector register file

General Purpose Registers (GPRs) are dynamically split between threads during kernel launch, allowing a trade-off between the number of registers per threads and the latency hiding capability. Barra maintains a separate state for each active warp in the multiprocessor. These states include a program counter, address and predicate registers, mask and address stacks, a window to the assigned register set, and a window to the shared memory.

4 Uniform and affine data in SPMD code

NVIDIA's so-called "scalar" architecture is actually a pure vector (SIMD) architecture. At the hardware level, all instructions, including memory and control-flow operations, operate on vectors, and all architecturally-visible registers are vectors. Though this provides a clean programming model by abstracting away the vector length and allows scalable implementations, it can become a source of inefficiency when performing inherently scalar operations. This issue is akin to value locality, where a correlation in time is observed in many values appearing in computations [6]. However in vector processors, correlations appear mostly inside vectors rather than between different time steps.

A uniform vector V is defined as having every component contain the same value $V_i = x$. Two main causes lead to *uniform* patterns. First, constant values and data read from memory with a uniform address vector (broadcast) generate uniform vectors. Second, uniform control flow is governed by uniform conditions. For example, a *for* loop with uniform bounds will also have a uniform counter. Modern GPUs also allow non-uniform conditions in conditional statements by allowing sub-vector control-flow. However, best performance is achieved when the control condition is uniform across a warp [1]. This means that in optimized algorithms, all lanes of registers that are used as conditions will hold the same value.

Similarly, to maximize memory bandwidth, memory accesses should follow specific patterns, such as the coalescing rules or conflict-free shared-memory access rules. In the Tesla architecture, memory addresses are usually computed using the regular SIMD ALUs. Programs following NVIDIA guidelines to access memory will operate mostly on consecutive addresses, which are common in vector programming. This correspond to a *affine* pattern, when threads access memory in sequence. In this case, the vector register V that stores the address is such that each component $V_i = x + iy$. One can notice that the uniform pattern is a specific case of the affine pattern when $y = 0$.

To quantify how often both of these patterns occur, we use Barra to dynamically check for each input and output operand in registers if they are uniform or affine vectors. We perform this analysis on two kinds of applications.

First, we used the examples from the CUDA SDK. Even though these examples are not initially meant to be used as benchmarks, they are currently the most standardized test suite of CUDA applications. As code examples, they reflect the best practices in CUDA programming.

The second benchmark is a bioinformatics application. RNAFold_GPU is a CUDA program which performs RNA folding. Based on dynamic programming, it achieves a 17-time speedup compared to a multicore implementation [7].

The proportion of uniform and affine inputs/outputs from and to registers is depicted in figure 3. Uniform or affine input data represent the percentage of uniform (affine) vectors among the data transferred between the register file and functional units.

Similarly, uniform or affine output data is the proportion of uniform (affine) data written back to the register file. It can be observed that whenever the

output is uniform (affine), the operation itself is executed on uniform (affine) data only.

We observe that a respective average of 27 % (44 %) of data read from the vector register file are uniform (affine) and 15 % (28 %) of data written back are uniform (affine). This proportion of uniform or affine inputs/outputs is significant enough to justify specific optimizations.

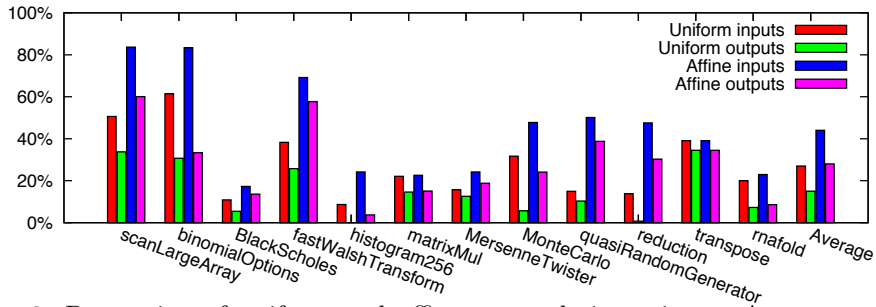


Fig. 3. Proportion of uniform and affine operands in registers. Averages are 27 % uniform inputs, 15 % uniform outputs, 44 % affine inputs and 28 % affine outputs.

5 Proposed technique

In this section we describe a technique which can detect if registers contain uniform or affine data as defined in Section 4. The first objective is to minimize memory and bus activity between the register file and the functional units for the proportion of input data captured by the proposed technique. The second objective extends the first one and goes further, by detecting uniform or affine data that are provided at the input of functional units and that remains uniform or affine at the output. In that case, the result can be computed by dedicated scalar hardware like in Cray processors or by relying on the existing vector hardware. As we target power reduction, the scalar solution would provide automatic reduction. However, this solution would cause data duplication in the register file and make the operand datapath more complex. The second solution can benefit from techniques that were not available at the time Cray machines were designed, such as clock-gating. This second solution can reuse the same vector hardware, with one or two scalar units enabled to compute the result, the other units being shut down using fine-grained stage-based clock gating as in the case of the IBM Cell SPU FPU [8]. The large vector length (32) used in the Tesla architecture promises larger power reductions than observed for more conventional SIMD extensions (typical length 4).

Instructions executed by GPUs show that most of uniform and scalar data come from a broadcast of some data or a copy of the register that contains the thread identifier. For these cases, uniform and scalar detection can be done statically for once at compile time or dynamically in hardware.

A static detection involves architectural as well as microarchitectural modifications. First, each instruction and register detected as uniform or scalar by the compiler has to be tagged in the instruction word. Then at runtime during the decode stage, the hardware can automatically schedule instructions according to the tag data. A dynamic detection keeps the instruction set unchanged. However, the burden of detecting uniform and scalar data is transfered from the compiler to the hardware. This solutions requires for example a tagged vector register file.

We tested the dynamic solution based on a tagged vector register file where each tag contains the type of data stored in the associated register (uniform, affine or generic vector) using the Barra simulator. At kernel launch time, the tag of the register that contains the thread identifier is set to the affine state. Instructions that broadcast values from a location in constant or shared memory set the tag of the result to the uniform state. Tags are then propagated across arithmetic instructions according to a simple set of rules, as shown in table 1. We arbitrarily restrict the allowed strides to powers of two to allow efficient hardware implementations, and conservatively make multiplications between affine and uniform data return vectors. Additionally, the information stored in this tag may be used by memory access units as it gives information about memory access patterns.

Table 1. Examples of rules of uniform and affine tag propagation. For each operation, the first row and first column indicate the tag of the first and second operand, respectively (Uniform, Affine or Vector). The central part contains the computed tag of the result.

+	U	A	V
U	U	A	V
A	A	V	V
V	V	V	V

×	U	A	V
U	U	V	V
A	V	V	V
V	V	V	V

<<	U	A	V
U	U	A	V
A	V	V	V
V	V	V	V

Cost. A tag array contains two bits per vector register. Each multiprocessor of a NVIDIA GT200 GPU features five hundred and twelve 1024-bit registers, for a total register-file size of 512 kb (not accounting for the size of error-correction codes, if any). In a basic implementation, the associated tags would require 1 kb of extra storage, making it comparatively almost negligible.

In terms of latency, reading the tags adds one level of indirection before reading registers. In NVIDIA GPUs, registers are read in sequence for a given instruction to minimize bank conflicts [9]. Therefore, operand reads can be pipelined with tag reads. Additionally, GPUs can tolerate large instruction latencies using fast context switching between threads. The tag of the output can then be computed using a few boolean operations from the tags of the input, so the required hardware modifications are minimal. Support for broadcasting a word across all SIMD units is already available to handle operands in Constant and Shared memory.

Benefits. When an input or output operand is known to be uniform, only one lane needs to be accessed. Likewise, affine vectors v such that $v_i = x + iy$ can be encoded using the base x and the stride y. Thus, their storage requirements are only two vector lanes. This reduces the used width of the register file ports and internal buses, thus saving power.

Computing a uniform or affine result function of uniform and affine inputs can be performed using only one or two Scalar Processing (SP) units with a throughput of one cycle instead of the full SIMD width during two cycles. Indeed, most arithmetic operations on affine vectors can be reduced to operations on the base and stride.

6 Technical issues

Some issues may limit the efficiency of the proposed method and need to be taken into account in an implementation.

Partial writes. GPUs handle branch divergence using predication. A predicated instruction does not write in every lane of its output register, keeping some of them in their previous state. In this case, even if the output value is uniform (or affine), the uniform (affine) property cannot be guaranteed for the destination register.

Half registers. The Tesla architecture allows access to lower/higher 16-bit sub-registers inside regular 32-bit registers. To handle this correctly, separate tags are needed for the lower, higher and whole parts to correctly track uniform/affine information.

Overflows. An arithmetic overflow may occur in a lane of an affine register, even if the base and stride are both representable. Overflows have no direct consequences when using two's-complement arithmetic, but casts between signed and unsigned formats of various sizes can occur, resulting for instance in an overflowing 16-bit affine value being extended into a non-affine 32-bit value.

This problem can be worked around by checking for overflows when performing affine computations, and re-issue the offending instruction as a vector operation when one is detected. Support for re-issuing instructions is already present to handle bank conflicts in the constant cache and scratchpad memory. As overflows should not occur in address calculations of correct programs, we expect it to be a rare occurrence. Indeed, we did not encounter this case in any of the benchmarks we ran.

Conversions from affine to generic vector. When an affine operand is combined with a vector operand, it needs to be first converted to a vector. As long as stride lengths are restricted to small powers of two, this can be implemented efficiently in hardware. However, it may be advantageous to reuse the conventional SIMD ALUs to perform the conversion, then re-issue the instruction if this situation is infrequent enough.

7 Results and validation

Figures 4 and 5 represent the respective proportions of uniform and affine operand captured with the proposed technique. We observe that on average, 19 % of inputs and 11 % of outputs can be identified as uniform data. These ratio go up to 34 % and 22 % respectively when considering affine data.

This means that the proposed methods reduce the bus activity between the register file and functional units for 34 % of the reads transfers. Likewise, the activity within the functional units can be reduced during 22 % of the operations executed in GPGPU computations. The power reduction brought by this technique, proportional to the activity reduction, is known to be of a critical issue for GPU [10]. Future works have to precisely quantify it.

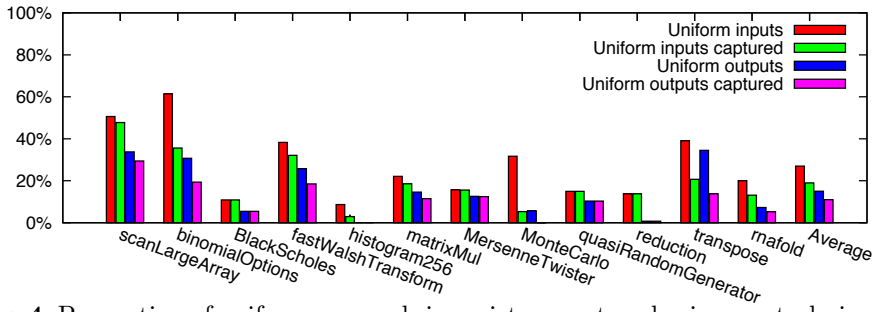


Fig. 4. Proportion of uniform operands in registers captured using our technique.

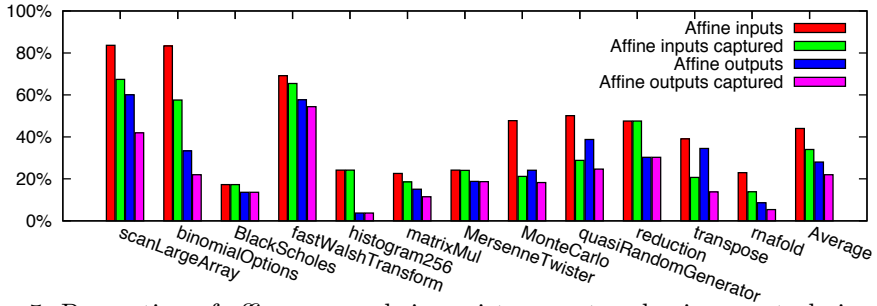


Fig. 5. Proportion of affine operands in registers captured using our technique.

It can be noted that the tag technique is not optimal, as it fails to detect some uniform and affine vectors. This is mostly due to the partial write effect as described in Section 6, and complex address calculations involving multiplication, division or modulo operations. Further work may improve the accuracy of the detection.

8 Conclusion

In this paper, we presented a technique to exploit two forms of value locality specific to vector computations encountered in GPUs. The first one corresponds

to the uniform pattern present when computing conditions which avoid divergence in sub-vectors. The second one corresponds to the affine pattern used to access memory efficiently. An analysis conducted on common programs used in the field of GPGPU showed that both of them are common. The novel idea of using both forms of value locality with the proposed modifications significantly reduces the power required for data transfers between the register file and the functional units as well as the power drawn by the SIMD arithmetic units. Future work will focus on improving the accuracy of the hardware-based dynamic technique presented in this article, as well as considering software-based static implementations.

Acknowledgments

We thank John Owens for his valuable comments on this work and Guillaume Rizk for the discussion related to bioinformatics applications. This work was partly supported by the French ANR BioWic.

References

1. NVIDIA: NVIDIA CUDA Compute Unified Device Architecture Programming Guide, Version 2.2. (2009)
2. Lindholm, J.E., Nickolls, J., Oberman, S., Montrym, J.: NVIDIA Tesla: A unified graphics and computing architecture. IEEE Micro **28**(2) (2008) 39–55
3. Bakhoda, A., Yuan, G., Fung, W.W.L., Wong, H., Aamodt, T.M.: Analyzing CUDA workloads using a detailed GPU simulator. In: proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), Boston (April 2009) 163–174
4. Diamos, G., Kerr, A., Kesavan, M.: Translating GPU binaries to tiered SIMD architectures with Ocelot. Technical Report GIT-CERCS-09-01, Georgia Institute of Technology (2009)
5. Collange, S., Defour, D., Parello, D.: Barra, a Modular Functional GPU Simulator for GPGPU. Technical Report hal-00359342, Université de Perpignan (2009) http://hal.archives-ouvertes.fr/hal-00359342/en/.
6. Balakrishnan, S., Sohi, G.S.: Exploiting value locality in physical register files. In: MICRO 36: Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture, Washington, DC, USA, IEEE Computer Society (2003) 265
7. Rizk, G., Lavenier, D.: GPU accelerated RNA folding algorithm. In: Computational Science – ICCS 2009. Volume 5544 of LNCS., Springer (2009) 1004–1013
8. Mueller, S., Jacobi, C., Oh, H.J., Tran, K., Cottier, S., Michael, B., Nishikawa, H., Totsuka, Y., Namatame, T., Yano, N., Machida, T., Dhong, S.: The vector floating-point unit in a synergistic processor element of a CELL processor. In: 17th IEEE Symposium on Computer Arithmetic (ARITH-17). (June 2005) 59–67
9. Lindholm, E., Siu, M.Y., Moy, S.S., Liu, S., Nickolls, J.R.: Simulating multiported memories using lower port count memories. US Patent US 7339592 B2 (March 2008) NVIDIA Corporation.
10. Collange, S., Defour, D., Tisserand, A.: Power Consumption of GPUs from a Software Perspective. In: ICCS 2009. Volume 5544 of Lecture Notes in Computer Science., Springer (2009) 922–931

Automatic Calibration of Performance Models on Heterogeneous Multicore Architectures

Cédric Augonnet, Samuel Thibault, and Raymond Namyst

INRIA Bordeaux, LaBRI, University of Bordeaux

Abstract. Multicore architectures featuring specialized accelerators are getting an increasing amount of attention, and this success will probably influence the design of future High Performance Computing hardware. Unfortunately, programmers are actually having a hard time trying to exploit all these heterogeneous computing units efficiently, and most existing efforts simply focus on providing tools to offload some computations on available accelerators. Recently, some runtime systems have been designed that exploit the idea of scheduling – as opposed to offloading – parallel tasks over the whole set of heterogeneous computing units. Scheduling tasks over heterogeneous platforms makes it necessary to use accurate prediction models in order to assign each task to its most adequate computing unit [2]. A deep knowledge of the application is usually required to model per-task performance models, based on the algorithmic complexity of the underlying numeric kernel.
We present an alternate, auto-tuning performance modelling approach based on performance history tables dynamically built during the application run. This approach does not require that the programmer provides some specific information. We show that, thanks to the use of a carefully chosen hash-function, our approach quickly achieves accurate performance estimations automatically. Our approach even outperforms regular *algorithmic* performance models with several linear algebra numerical kernels.

1 Introduction

Multicore architectures are now widely adopted throughout the computer ecosystem. There are also clear evidences that solutions based on specialized hardware, such as accelerator devices (*e.g.* GPGPUs) or integrated coprocessors (*e.g.* Cell's SPUs) are offering promising answers to the physical limits met by processor designers. Future processors will therefore not only get more cores, but some of them will be tailored for specific workloads.

In spite of their promising performance in terms of computational capabilities and power efficiency, such heterogeneous multicore architectures require appropriate tools. This introduces challenging problems at all levels, ranging from programming models and compilers to the design of libraries with a real support for heterogeneity. As they offer dynamic support for what has become hardly doable in a static fashion, runtime systems have a central role in this software

stack. In previous work, we have therefore developed StarPU [2], a unified runtime system that offers support for heterogeneous multicore architectures. Its specificity is that it not only targets accelerators (GPUs, Cell's SPUs, etc.) but also multicore processors *at the same time*, in a portable fashion. StarPU also provides portable performance thanks to a high-level framework for designing portable scheduling policies.

Performance modeling is a very common technique in the scheduling literature. Whenever doable, practically building such models usually requires consequent efforts along with a certain knowledge of both the application algorithm and the underlying architecture. This is even more difficult in the case of heterogeneous platforms. But without an appropriate interface, such knowledge is not available from the runtime systems' perspective: describing a task as a function pointer and pointers to the data (similar to OpenMP 3.0 tasks) does not really give much information to the runtime system in charge of the scheduling. (Un)fortunately, current accelerators reintroduces the problem of data management across distributed a memory model, so that we *have* to adopt much more expressive task APIs anyway. The majority of the programming models that target accelerators (and that do not just delegate data movements to the programmers!) require to explicitly describe which data is accessed by a task [6,7,10,1]. While this adds constraints on the programmers who have to adapt their applications to those expressive programming interfaces, the underlying runtime system gets much more information.

In this paper, we explain how StarPU takes advantage of that expressiveness to seamlessly build performance models on heterogeneous multicore architectures. Then, we illustrate how this systematic approach performs in terms of prediction accuracy and regarding its impact on the actual performance. Finally, we show that StarPU not only grabs information from the programming interface to perform better scheduling, but it also returns performance feedback information thanks to convenient tools which are helpful for instance in the context of auto-tuned libraries or when analyzing performance.

2 StarPU, a runtime system for heterogeneous machines

In this section, we briefly present StarPU, our unified runtime system designed for heterogeneous multicore platforms, described in more details in a previous paper [2]. It distributes tasks onto both accelerators and processors *simultaneously* while offering portable performance thanks to generic scheduling facilities.

2.1 A unified runtime system

The design of StarPU is organized around three main components: a portable offloable-task abstraction, a library that manages data movements across heterogeneous platforms, and a flexible framework to design portable scheduling policies.

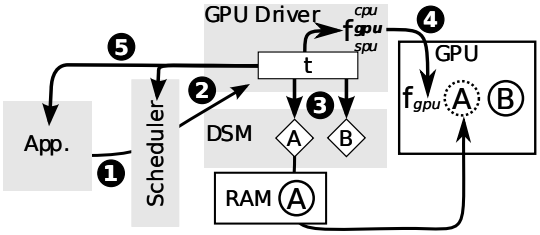


Fig. 1. Execution of a Task within StarPU. Applications submit tasks that are dispatched onto the different drivers by the scheduler. The driver offloads the computation, using the proper implementation from the codelet, and the DSM ensures the availability of coherent data. A callback is executed when the task is done.

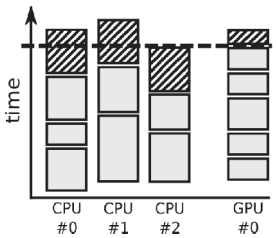


Fig. 2. The "Earliest Finish" Scheduling Strategy.

A unified execution model. StarPU introduces the notion of *codelet*, which is the set of implementations of the same computation kernel (*e.g.* a vector sum) for different computation units (*e.g.* CPU and GPU). A StarPU task is then an instance of a codelet applied to some data. Figure 1 shows the path followed by tasks in StarPU. Programmers submit (graphs of) tasks to StarPU which is responsible for mapping them as efficiently as possible on the eligible processing units. Instead of hard-coding all the interactions between the processing units, StarPU makes it possible to concentrate on the design of efficient computational kernels and algorithmic problems instead of being stuck by low-level concerns.

A data management library. Maintaining data coherency (and availability) is a crucial issue with accelerators. In a previous paper [1], we have designed a high-level data management library that is integrated in StarPU. Mapping data statically is not necessarily sufficient when multiple processing units access the same pieces of data. The resulting data transfers are critical for the overall performance so that integrating data management within StarPU made it possible to apply optimizations (*e.g.* prefetching, reordering, asynchronous memory transfers) and to guide the scheduler.

A scheduling framework. StarPU not only executes tasks, but it also maps them as efficiently as possible thanks to its expressive scheduling interface. Hence, StarPU offers a flexible framework to implement portable scheduling policies [2]. Such policies are portable in the sense that they are directly applicable to platforms as different as a Cell processor and a hybrid GPU/CPUs machine.

2.2 Scheduling strategies based on performance models

In a previous paper [2], we have presented various scheduling policies implemented in StarPU with relatively few efforts. For instance, one of these policies is similar to the HEFT scheduling strategy [12]. As shown on Figure 2, the scheduler keeps track of the expected duration until the different processing units are

available. When a task is submitted to the scheduler, it is attributed to the processing unit that minimizes termination time according to the expected duration of the task on the different architectures (depicted by hatchings).

We have for instance used this rather simple strategy successfully to obtain superlinear speedups on an LU decomposition thanks to per-architecture performance models that take into account the (lack of) affinity of tasks with the different processing units. However this strategy requires that we can approximate the execution time of the tasks on the various architectures.

3 Dynamically building Performance Models

In this section, we discuss how we can build performance models, and we give a systematic approach to dynamically construct and query a performance model based on historical knowledge, seamlessly for the programmers.

In the context of dynamic task scheduling, we do not need perfectly accurate models, but we need to take appropriate decisions when assigning the tasks onto the different processing units. Our performance models should for instance capture the relative speedups as well as the affinities between tasks and processors.

3.1 How to define a performance model?

In order to define a performance model, we need to decide which parameters the model should depend on.

The most obvious parameters to describe a task are the kernel and the architecture: in the case of a matrix product on CUDA, we could for instance identify a task by the pair (SGEMM, CUDA) and associate it with its predicted execution time. A trivial refinement is to consider the total size of the tasks' data, so we can also associate this pair with a parametric cost function depending on that size (*e.g.* $\mathcal{C}(S) = \alpha S^{3/2}$ in the case of SGEMM, α being a parameter that has to be defined).

The total size is often not sufficient: in the case of a kernel handling a $(n \times m)$ matrix in $\mathcal{O}(n^2 m)$, we must make a difference between (1024×512) and (512×1024) matrices. Such multivariate models are however only applicable if we have sufficient knowledge of the algorithm, which a runtime system could hardly infer automatically in a generic way. Finding an explicit model of the execution time can also be awkward because of architectural concerns such as the size of the caches. Using piecewise models is possible, but it requires to delimit the boundaries of the pieces, which can be time demanding, especially for a multivariate model and in a heterogeneous environment.

In many classes of algorithms, we can reasonably make some extra regularity assumption such as most tasks handling blocks of fixed size (*e.g.* in tiled algorithms), or a limited set of sizes (*e.g.* in divide and conquer algorithms). In this case, explicitly modeling the performance as a function of the data size can be unnecessarily complicated. A history-based approach would be much simpler: instead of using a complicated multivariate model to differentiate between a

(1024×512) matrix and a (512×1024) one, we simply store the execution time that was measured for those different input configurations. The advantage of this approach is that it is transparent to the programmer as long we have some mechanism to match a task with those previously executed. In the next section, we present how StarPU implements history-based models in a flexible way, with sufficient performance feedback to help programmers easily decide whether this is an appropriate model or not.

3.2 How to build performance models?

There are various ways to determine the parameters required to build the performance models that we have described in the previous section: either completely manual, or completely automated, depending on the type of the adopted model.

Building a performance model by hand (*e.g.* using the ratio between the number of operations and the speed of the processor) is hardly applicable to modern processors and require a perfect knowledge of both the application and the architecture. In the case of heterogeneous multicore processors, with multiple processing units to handle, this becomes rather unrealistic. It is however possible to design a model based on the amount of computations per task, and to calibrate the parameters by the means of a regression.

It is common to use specific precalibration programs to build those models. While this may be suited for kernels that are widely used (*e.g.* BLAS), this requires a specific test-suite and the corresponding inputs, which often represents an important programming overhead. In the context of multicore architectures, it is even harder to create a realistic workload: independently benchmarking the various processing units without taking into account the various interactions (*e.g.* cache sharing or bus contention) may not result in reliable measures.

On the other hand, it is possible to measure the performance of the different tasks during an actual execution. This does not require any additional programs, and it provides realistic performance measurements. StarPU can therefore automatically calibrate parametric models, either at runtime using linear regression models (*e.g.* $\mathcal{C}(n) = \mathcal{O}(n^\alpha)$) or offline in the case of non-linear models (*e.g.* $\mathcal{C}(n) = \alpha n^\beta + \gamma$, as shown on Figure 7). StarPU also builds history-based performance models by storing the performance of the tasks on the different inputs, transparently for the application.

3.3 A generic approach for building history-based performance models dynamically

This section shows how StarPU keeps track of the performance obtained by the tasks on the different input, and how it is possible to match a task with its similar predecessors. As shown on Figure 3, this process involves three main steps: measuring the actual duration of the tasks when they are executed and integrating this measurements in the history log of the task; being able to look-up the performance of some task according to the previous measurements; and offering some performance feedback to the application.

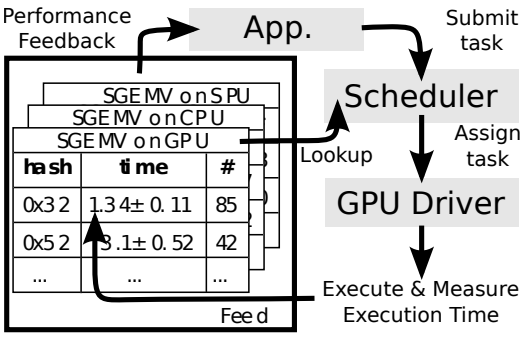


Fig. 3. Performance feedback loop.

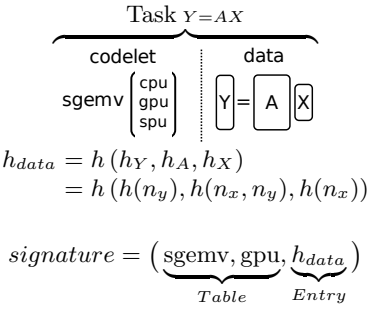


Fig. 4. Uniquely identifying a task

Measuring tasks' duration. Measuring the time spent to compute a task is usually simple thanks to the cycle counter facility provided by most constructors. In the case of Cell processors, we had to make the SPUs transmit those measurements to the PPU along with the output data, but this is not an intrusive mechanism since those DMA transfers are overlapped.

Identifying task kinds. We use the layout and size of the data to distinguish the different kind of instances of a computational kernel. We now present how to compute a hash value to characterize the data layout of a task.

StarPU's data management library not only manipulates buffers described by a pointer and its length, but it also handles a mixture of various high-level data *interfaces* [1]. On Figure 4, a matrix-vector product accesses a set of matrices and vectors. There can also be much more complex data interfaces (*e.g.* compressed sparse matrices), but the size of any piece of data can be characterized by a k-uplet of parameters $(p_1, \ldots, p_k)$ where k and the parameters depend only on the data interface. A matrix is for instance described by a pair (n, m), and a single parameter is sufficient to describe the length of a vector.

We now define a **hash** function that computes a unique identifier for such a set of parameters. As shown on Figure 4, we characterize the size of each piece of data by applying a hash function [1] to the parameters $p_1, \ldots, p_{k-1}$ describing it. By then applying the hash function to the different per-data hashes, we get a characterization of the data layout and size for the whole task. Applying this method on a tiled algorithm would for instance result in having as many hash values as there are tile sizes.

Feeding and looking up from the model. It is now extremely simple to implement a model based on the history in StarPU: each computational kernel is associated with a hash table per architecture. When a task is submitted to StarPU, it computes its hash, and consults the hash table corresponding to the proper kernel-architecture pair to retrieve the average execution time previously measured for this kind of task. The average execution time and other metrics

[1] For example, we can use the usual CRC hash functions: $h(p_1, \ldots, p_k) = CRC(p_1, \ldots, CRC(p_{k-1}, CRC(p_k, 0)))$

such as standard deviation are updated when a new measure is available. Hash tables can be saved (or loaded) to (from) a file so that these performance models are persistent between different runs. It is therefore possible to rapidly calibrate models by running small problems that have the same granularity as the actual problems.

4 Experimental validation

We have implemented these automatic model calibration mechanisms in StarPU which runs on multicore CPUs, GPUs and Cell processors. In this section, we give evidence that they have a significant impact on performance; we also illustrate the performance feedback offered by StarPU, and how StarPU provides some tools to help programmers understand the performance they obtain, and to select the most appropriate models in consequence. We here show how these mechanisms perform in the case of a hybrid platform with a nVidia Quadro FX4600 GPU and a E5410 Xeon quad-core CPU.

4.1 Sharpness of the performance prediction

Figure 5 shows the results obtained on an LU decomposition for two different problem sizes. The first line exhibits the average and standard deviation of the reference performance obtained when using a greedy scheduling policy to distribute tasks to CPUs and the GPU. The second line shows the results obtained when calibrating the history-based performance model after either one, two or three runs and the average performance (and standard deviation) obtained after 4 runs. During the first execution, the greedy strategy clearly outperforms the non-calibrated strategy based on performance models. But once the model is calibrated, the performance obtained by the model-based strategy gets better, not only in terms of average speed, but also with respect to the standard deviation. The improvement between the runs is explained by the fact that the application runs on a hybrid CPU/GPU platform: the better the accuracy, the better the

	Speed (GFlop/s)	
Policy \ Size	$(16k \times 16k)$	$(30k \times 30k)$
Greedy (avg.)	89.98 ± 2.97	130.68 ± 1.66
Perf. Model 1^{st} iter.	48.31	96.63
Perf. Model 2^{nd} iter.	103.62	130.23
Perf. Model 3^{rd} iter.	103.11	133.50
Perf. Model ≥ 4 (avg.)	103.92 ± 0.46	135.90 ± 0.64

Fig. 5. Impact of performance sampling on the speed of an LU decomposition (in GFlop/s)

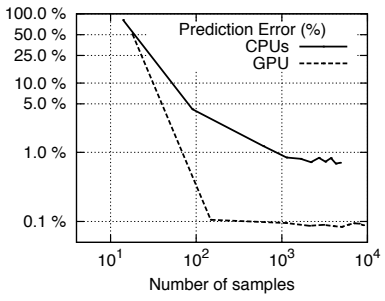


Fig. 6. Performance model accuracy

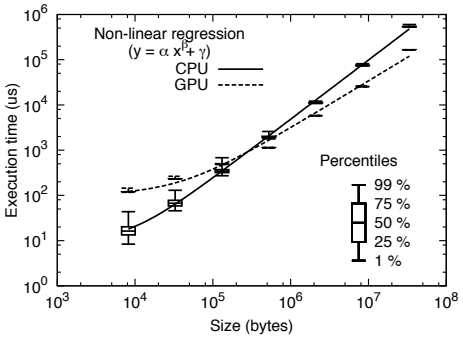


Fig. 7. Performance and regularity of an STRSM BLAS3 kernel depending on granularity.

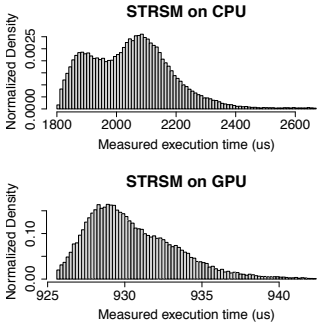


Fig. 8. Distribution of the execution times of a STRSM kernel measured for tiles of size (512×512).

load balancing. Until the models are properly calibrated, some processing units receive too much work while others are not kept busy enough.

Figure 6 depicts the evolution of the prediction inaccuracies depending on the number of collected samples. More precisely, the error is computed by taking the sum of the absolute differences between prediction and measurements, for all tasks, and by dividing this total prediction error by the total execution time. As suggested by Figure 5, the accuracy of the models becomes better as we keep collecting measurements. We finally obtain an accuracy of an order of 1% for multicore CPUs, and below 0.1% for a GPU. This difference is due to complex interactions occuring within multicore CPUs (*e.g.* cache sharing and contention) while computations are not perturbed on GPUs. The large majority of tasks in an LU decomposition are matrix products, whose performance is especially regular even on CPUs, so that we obtain a relatively good overall accuracy.

4.2 Performance feedback tools

StarPU provides tools to detect tasks that are not predictable enough (*e.g.* BLAS1 kernels). Figures 7 and 8 are automatically generated by StarPU, which can collect performance measurements at runtime.

Figure 7 summarizes the behaviour of a kernel on all input sizes and the performance variations observed for the different sizes, and for the different architectures; it also shows the non-linear regression-based performance models automatically generated by StarPU so that we can figure out whether such a model is applicable or not. It also illustrates in which situation it is worth using accelerators or CPUs, therefore helping to select the most appropriate granularity. Using a small grain size on CPUs results in variable execution times, certainly explained by a poor cache use which makes performance very sensitive to the bus contention for instance. This problem disappears as we take large tiles, or if we use a GPU that is much less sensitive to such variations.

Figure 8 shows the actual distribution of the measurements that were collected for a given hash value. This not only gives a precise idea of the performance dispersion, but it can also be used to understand the actual performance issues: on the very predictable GPUs, we obtain a very thin peak, while on the CPUs, the distribution exhibiting two hills suggests that there may be some contention issue which should be further analyzed.

5 Related Works

Auto-tuning techniques have been successfully used to automatically generate the kernels of various high-performance libraries such as ATLAS [4], FFTW, OSKI or SPIRAL; and similar results are obtained in the context of GPU computing by the MAGMA project[9]. While performance models permit to generate efficient computational kernels even on heterogeneous systems, computations are usually mapped statically on the different processing resources when dealing with hybrid systems [11].

Iterative compilation frameworks also use performance feedback to take the most appropriate optimization decisions. Jimenez *et al.* [8] keep track of the relative speedups of the applications on the different architectures to decide which processing unit should be assigned to an application. Their approach is much less flexible since it does not allow to actually schedule interdependent tasks within an application.

Different runtime systems currently offer support for accelerators [3], or even hybrid systems. Similarly to StarPU, the Harmony runtime system targets hybrid platforms while proposing some scheduling facilities, possibly based on performance modeling [5]. Its performance is modeled by the means of (possibly multivariate) regression models. This approach is hardly applicable without any support from the programmer, and possibly requires a large number of samples to have a reliable model. Thanks to the high-level support for data management integrated within StarPU, the history-based solution that we propose in this paper is simpler as it is completely transparent for the programmers.

6 Conclusion

We have proposed a generic approach to seamlessly build history-based performance models. It has been implemented within the StarPU runtime system with the support of its integrated data management library, and we have shown how StarPU's performance feedback tools help programmers analyze whether the resulting performance prediction are relevant or not.

Such history-based performance models naturally rely on some regularity hypothesis since it cannot predict the behaviour of a task if all its predecessors had different sizes: in that case, a parametric performance model calibrated by the means of regressions is more suitable. Our history-based approach also requires computational kernel with a static flow control. Tasks' execution time should be independent from the actual content of the data, the latter is often

unknown when the scheduling decisions are taken anyway. Since our approach does not require any effort from the programmers, they can easily use our automatic calibration mechanisms to see whether the use of such models results into performance improvements.

This technique is directly applicable to the case of complex hybrid setups (*e.g.* heterogeneous multiGPU). This work could also be extended to model the performance of memory transfers so that StarPU could schedule them as well. Scheduling policies could take advantage of performance models that depend on the actual state of the underlying machine: using hardware performance counters, the history-based models could for instance keep track of contention or cache usage. Finally, performance feedback can be valuable: this not only helps to understand the behaviour of an application during a post-mortem analysis, but this is also useful for iterative compilation environments and auto-tuned libraries.

References

1. C. Augonnet and R. Namyst. A unified runtime system for heterogeneous multicore architectures. In *Euro-Par 2008 Workshops - HPPC'08*, Las Palmas de Gran Canaria, Spain, August 2008.
2. C. Augonnet, S. Thibault, R. Namyst, and P.A. Wacrenier. StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures. In *Proceedings of the 15th Euro-Par Conference*, Delft, The Netherlands, August 2009.
3. P. Bellens, J.M. Perez, R. M. Badia, and J. Labarta. Cellss: a programming model for the cell be architecture. In *SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, page 86, New York, NY, USA, 2006. ACM.
4. R. Clint Whaley and J. Dongarra. Automatically Tuned Linear Algebra Software. In *Ninth SIAM Conference on Parallel Processing for Scientific Computing*, 1999.
5. G. Diamos and S. Yalamanchili. Harmony: Runtime Techniques for Dynamic Concurrency Inference, Resource Constrained Hierarchical Scheduling, and Online Optimization in Heterogeneous Multiprocessor Systems. Technical report, Georgia Institute of Technology, Computer Architecture and Systems Lab, 2008.
6. A. Duran, J.M. Perez, E. Ayguade, R. Badia, and J. Labarta. Extending the openmp tasking model to allow dependant tasks. In *IWOMP Proceedings*, 2008.
7. K. Fatahalian, T.J. Knight, M. Houston, M. Erez, D. Reiter Horn, L. Leem, J. Young Park, M. Ren, A. Aiken, W.J. Dally, and P. Hanrahan. Sequoia: Programming the memory hierarchy. In *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, 2006.
8. V.J. Jiménez, L. Vilanova, I. Gelado, M. Gil, G. Fursin, and N. Navarro. Predictive runtime code scheduling for heterogeneous architectures. In *HiPEAC*, 2009.
9. Y. Li, J. Dongarra, and S. Tomov. A Note on Auto-tuning GEMM for GPUs. In *ICCS (1)*, pages 884–892, 2009.
10. M.D. McCool. Data-Parallel Programming on the Cell BE and the GPU using the RapidMind Development Platform. In *GSPx'06 Multicore Applications Conference*.
11. S. Tomov, J. Dongarra, and M. Baboulin. Towards Dense Linear Algebra for Hybrid GPU Accelerated Manycore Systems. Technical report, January 2009.
12. H. Topcuoglu, S. Hariri, and Min-You Wu. Performance-effective and low-complexity task scheduling for heterogeneous computing. *Parallel and Distributed Systems, IEEE Transactions on*, 13(3):260–274, Mar 2002.

KEYNOTE

Software Development and programming of multicore SoC

Ahmed Jerraya, CEA-LETI, MINATEC, France

Abstract: SoC designs integrate an increasing number of heterogeneous programmable units (CPU, ASIP and DSP subsystems) and sophisticated communication interconnects. In conventional computers programming is based on an operating system that fully hide the underlying hardware architecture. Unlike classic computers, the design of SoC includes the building of application specific memory architecture and specific interconnect and other kinds of hardware components required to efficiently executing the software for a well defined class of applications. In this case, the programming model hides both hardware and software interfaces that may include sophisticated communication and synchronization concepts to handle parallel programs running on the processors. When the processors are heterogeneous, multiple software stacks may be required. Additionally, when specific Hardware peripherals are used, the development of Hardware dependent Software (HdS) requires a long, fastidious and error prone development and debug cycle. This talk deals with challenges and opportunities for the design and programming of such complex devices.

Bio: *Dr. Ahmed Jerraya is Director of Strategic Design Programs at CEA/LETI France. He served as General Chair for the Conference DATE in 2001, Co-founded MPSoC Forum (Multiprocessor system on chip) and is the organization chair of ESWEEK2009. He supervised 51 PhD, co authored 8 Books and published more than 250 papers in International Conferences and Journals.*

PANEL

Are many-core computer vendors on track?

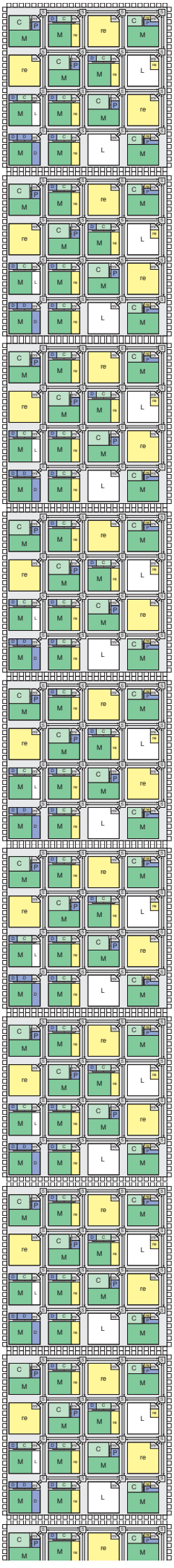
Martti Forsell, VTT, Finland
Peter Hofstee, IBM Systems and Technology Group, USA
Ahmed Jerraya, CEA-LETI, MINATEC, France
Chris Jesshope, University of Amsterdam, the Netherlands
Uzi Vishkin, University of Maryland, USA

Moderator: Jesper Larsson Träff, NEC Laboratories Europe, Germany

Outline: The current proliferation of (highly) parallel many-core architectures (homo- and heterogeneous CMP's, GPU's, accellerators) puts an extreme burden on the programmer seeking (or forced) to effectively, efficiently, and with reasonable portability guarantees utilize such devices. The panel will consider whether what many-core vendors are doing now will get us to scalable machines that can be effectively programmed for parallelism by a broad group of users.

Issues that may be addressed by the panelists include (but not exclusively): Will a typical CS graduate be able to program main-stream, projected many-core architectures? Is there a road to portability between different types of many-core architectures? If not, should the major vendors look for other, perhaps more innovative, approaches to (highly) parallel many-core architectures? What characteristics should such many-core architectures have? Can programming models, parallel languages, libraries, and other software help? Is parallel processing research on track? What will the typical CS student need in the coming years?

the 3rd Workshop on Highly Parallel Processing on a Chip



VTT NEC Euro-Par