Device specialization in heterogeneous multi-GPU environments

Gabriele Cocco[1] **and Antonio Cisternino**[1]

1 **Computer Science Dept., University of Pisa**
Largo Bruno Pontecorvo, Pisa, Italy
`cocco@di.unipi.it, cisterni@di.unipi.it`

Abstract

In the last few years there have been many activities towards coupling CPUs and GPUs in order to get the most from CPU-GPU heterogeneous systems. One of the main problems that prevent these systems to be exploited in a device-aware manner is the CPU-GPU communication bottleneck, which often doesn't allow to produce code more efficient than the GPU-only and the CPU-only counterparts. As a consequence, most of the heterogeneous scheduling systems treat CPUs and GPUs as homogeneous nodes, electing map-like data partitioning to employ both these processing resources. We propose to study how the radical change in the connection between GPU, CPU and memory characterizing the APUs (Accelerated Processing Units) affect the architecture of a compiler and if it is possible to use all these computing resources in a device-aware manner. We investigate on a methodology to analyze the devices that populate heterogeneous multi-GPU systems and to classify general purpose algorithms in order to perform near-optimal control flow and data partitioning.

1998 ACM Subject Classification C.1.3 Other Architecture Styles, D.1.3 Concurrent Programming, D.2.8 Metrics

Keywords and phrases HPC, APU, GPU, GPGPU, Heterogeneous computing, Parallel computing, Task scheduling

Digital Object Identifier 10.4230/OASIcs.ICCSW.2012.35

1 Introduction

In the last few years various researches demonstrated that GPU computing power isn't suitable to accelerate many algorithms[4], mostly due to the execution model and to the limited performances of the interconnection between the GPUs and the rest of the system. From the execution model point of view, SIMD doesn't fit very well with some computations, such as algorithms containing many branches and fine-grained data-parallel computations. Among the algorithms falling into these categories we can find Huffman Coding[6, 7] and KD trees construction[8]. To get the highest performances, the characteristics of the CPUs, such as wide caches and Multiple Thread Multiple Data (MTMD) execution model, should be employed to accelerate portions of such kind of algorithms. Unfortunately, partitioning algorithms to run heterogeneously on CPU-GPU systems is hold by the CPU-GPU interconnection and communication performance[2]. Whereas an algorithm may benefit to be carefully staged across the two computing resources, the time spent in transferring data often outweighs the time saved in executing code on the most specific resource. Given this, recent researches have mostly focused on exploiting the CPU and the GPU in an homogeneous way, ignoring the specific characteristics of each computing resource and partitioning data in a task-farm manner instead of scheduling differents parts of the control flow. S. Venkatasubramanian and R. W. Vuduc [9] propose a solution to employ heterogeneous CPU-GPU platforms to accelerate Jacobi's iterative method for 2D Poisson equations. Since the target of this work is quite specific, hand-crafted implementations for CPU and GPU are proposed instead of a methodology to partition more general stencil algorithms and to automatically produce target code. C. Luk and S.H.H. Kim [11] present an entire programming system for CPU-GPU platforms where mapping between tasks and processing resources can be either performed by the programmer or automatically by the scheduler. While manual mapping may allow to partition control flow in addition to data, it substatially charges the programmer of determining a good partitioning strategy and hand-coding the implementations for both CPU and GPU. In [10], a machine learning approach is employed to statically decide a near-optimal scheduling strategy. Like in the other works, partitioning is based only on data and predictors are used to determine the amount of data to schedule to each processing element instead of which part of the algorithm to execute on it. In [14] a performance-history based scheduler is proposed to schedule tasks on the computing resource that demonstrated to be the most efficient in executing those tasks during previous executions. Since tasks are not analyzed nor classified on the bases of their features but are instead considered as blackboxes of which only the completion time is known, the scheduler requires to be retrained every time it is ported to a different CPU-GPU platform. E. Hermann et al. present a task scheduling approach for interactive physics simulations that allows to split tasks across multiple CPUs and GPUs[12] on the basis of task size and estimated completion time.

With the introduction of APUs, such as the Intel Ivy Bridge® and the AMD Fusion® family, the CPU-GPU communication performance has decisively increased[1], thanks to a novel architectural interconnection that overcomes the limits of the PCI-express bus and to the chance for the CPU and the GPU to effectively share data without the need for copies. APUs may therefore raise the chance for algorithms to be partitioned across heterogeneous processing resources in a device-specific manner. At the same time, given the different balances between computing and communication performance of integrated versus discrete GPUs, APU's and discrete GPUs in a hybrid multi-GPU system can be indepedently specialized to accelerate different kind of computations.

Our work consists in leveraging on the APU's capabilities to investigate on the chances to specialize both the on-chip resources and discrete GPUs in order to schedule portions of data and control flow on the bases of the specific characteristics of each device. Since APUs represent a relatively recent architecture, there are currently few resources on them, such as costs models, performance analysis or researches towards scheduling strategies to exploit this kind of tightly coupled platforms. S. Keely[8] discusses about adaptive mapping kd-tree construction on APUs to get the most from both the CPU and the integrated GPU, showing that on-chip communication performance allows to reduce the penalities of synchronization and host-device data transfer costs. K. Spafford et al.[5] propose an extensive comparison of various algorithms running on discrete versus APU's GPUs, illustrating the benefits of tighter coupling when data exchange becomes a dominant portion of the runtime. M. Daga at al. show a similar comparison[1], but int this case the intergrated GPU and a discrete one are tested using two different platforms. Since the two GPUs do not share the CPU executing the host code, comparing their performances is difficult and not fully reliable.

2 Methodology

The main aspects that characterize APUs is an high CPU-GPU communication performance and a computing power usually much lower than the computing power of mainstream discrete

GPUs. Given this, computations for which the time spent in CPU-GPU communication outclasses computing time should benefit of integrated GPU execution. The first step of our work consists in verifying this idea using a set of algorithms with different data-transfer/operations ratio (section 3.1).

The second step of the research consists in defining a reliable metric for classification of algorithms in terms of the device (integrated or discrete GPU) that is more suitable for their execution. We plan to train this metric on a particular system using a set of samples (i.e. popular and widely used parallel programming patterns) and taking into account the characteristics of the various devices in order to obtain a convenience threshold. Algorithms for which the value of the metric is below the threshold should be scheduled on the APU's GPU while algorithms for which the metric is above the threshold should be run on the discrete GPU. Following the results of the first step, the classification metric should be mainly based on the amount of data transferred and on the amount of operations executed on data. The term "operations" may refer to device ISA or intermediate language (PTX, AMD IL) instructions, to higher-level assembly instructions (MSIL, LLVM), to source-language-based operations (C/C++ assignments, arithmetic operations, etc.) or to language-unaware algorithmic complexity. Even if moving from device ISA to virtual machine assembly affects the amount of instructions executed, the number of operations considered in the metric should not necessarly match instructions executed by a device but instead characterize the complexity of the algorithm in a device-unaware manner. Moreover, as shown in section 3, the relative integrated versus discrete GPU performance seems to be correlated to the complexity of the (sequential) algorithm, which suggests the reliability of expressing operations in terms of sequential algorithms. The first definition of this metric, based on language-unaware algorithmic complexity, is discussed in section (3.2). For future work we plan to move to LLVM and to analyze LLVM code to determine the value of the metric for generic algorithms. LLVM provides various tools and services, such as loop informations, dead-code elimination and more, that can be exploited to simplify code analysis.

In addition to the operations executed and to the amount of data transferred other aspects may affect the GPUs relative performance, such as synchronization, branching, number of working threads and memory conflicts. While most of these aspects might be taken into account, we decide to start with a very simple metric and to refine it introducing additional parameters as soon as we encounter examples for which the metric fails in modeling the integrated versus discrete GPU performance. In addition, some of these features, like synchronizations and memory conflicts, are not exposed at LLVM level. Whenever the metric requires to take them into account, an OpenCL implementation of the algorithm should be available or it should be generated starting from LLVM implementation. AMD is currently developing an LLVM backend to produce AMD IL binaries, while an LLVM to Nvidia PTX is already available [19].

Founding our work on a device-unaware analysis of operations allows to overcome the dependency of device ISA/IL from the specific GPU model and vendor and of the source code from a specific programming language. Moreover, working on LLVM allows to extend the analysis to include the CPU as a scheduling target.

Since the final target of our research is to partition and to schedule generic algorithms in a control-flow-aware manner, the most important step consists in developing a partitioning and scheduling system based on code similarity pattern discovery engine to recognize well-known parallel patterns inside generic algorithms. The engine is paired with a database of popular computations, such as map, reduce, scan, convolution, matrix reduction and multiplication. The classification metric defined in the previous steps is used to train the scheduler on a

Table 1 Specifications of the testing platform.

Device	Clock rate	SIMDs	Cores	Processing power
AMD A8-3850 (CPU)	2.9 GHz	-	4 CPUs	-
AMD 6550D (Integrated)	600 MHz	5	400 radeon	480 GFLOPS
AMD HD 5870 (Discrete)	850 MHz	20	1600 radeon	2720 GFLOPS

particular system to determine the integrated versus discrete GPU convenience threshold. For each computation in the database we store a marker representing the device on which it is more suitable to execute. Finally, the database and the markers are employed to partition a generic algorithm in terms of the patterns it contains, scheduling each pattern recognized on the device stored in the relative marker. For this part of the work we can benefit of many researches on code similarity and pattern discovery [16, 17, 18].

3 Preliminary results

In order to investigate on the chance for the CPU, the integrated and the discrete GPU to be specialized to accelerate different sets of computations, we start analyzing the performances of integrated and discrete GPUs in running a set of algorithms with specific computing requirements. The algorithms that compose the test suite are vector/matrix addition (saxpy), reduction, convolution and matrix multiplication. The choice of the set of algorithms has been driven by the aim to take into account both memory-bound and compute-bound algorithms.

Each algorithm is executed on the testing platform (table 1) under different conditions. In particular, we run each algorithm using all the possible data-transfer strategies, like mapping, placement (ALLOC_HOST_PTR, USE_PERSISTENT_MEM_AMD, etc.) and pre-pinning and employing different data types (float, float2, float4, etc.). For each algorithm and for each device we select the conditions that lead to the lowest completion time.

3.1 Experimental results

Figures 1, 2 and 3 compare the completion times of the integrated GPU and of the discrete GPU resulting from the execution of the benchmarks. Since we show the ratio between the completion times of the two devices, values higher than one mean that the integrated GPU is faster than the discrete GPU, while a ratio lower than one signifies that the discrete GPU is faster than the integrated one.

The integrated GPU is more efficient in executing both saxpy and reduction for every input size. Matrix multiplication falls into the opposite situation, since the discrete GPU outperforms the integrated one regardless the matrix size. Finally, convolution exhibits a mixed behaviour, where the discrete GPU is faster for small input matrixes and gradually becomes slower than the integrated GPU as bigger the matrix size. Since the convolution algorithm depends on both the input matrix size and the filter size, we also run the algorithm fixing the input matrix size and gradually increasing the filter. The results of this test, shown in figure 4, confirm the partial convenience of the integrated GPU and allow to conclude that the discrete GPU outperforms the integrated one for small input matrixes and for big filters.

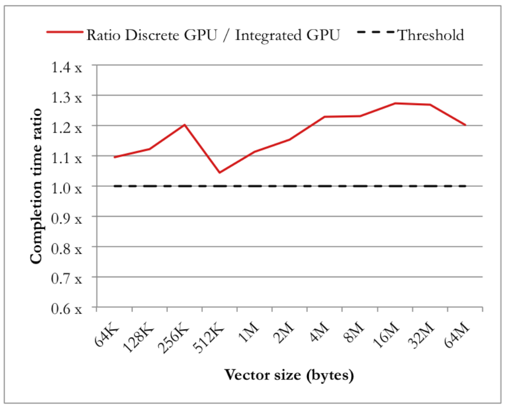

Figure 1 Saxpy completion time ratio: discrete GPU / integrated GPU.

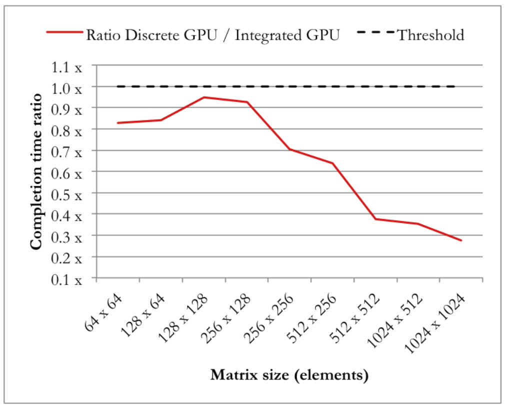

Figure 2 Matrix multiplication completion time ratio: discrete / integrated GPU.

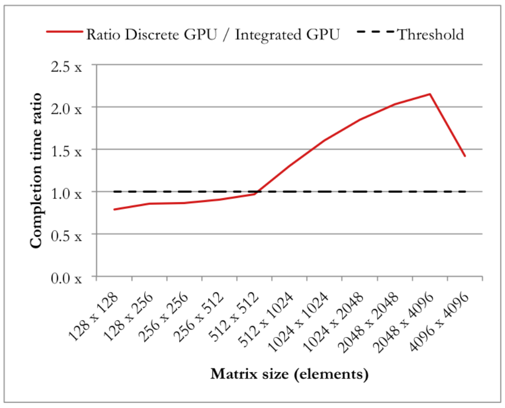

Figure 3 Convolution completion time ratio: discrete / integrated GPU.

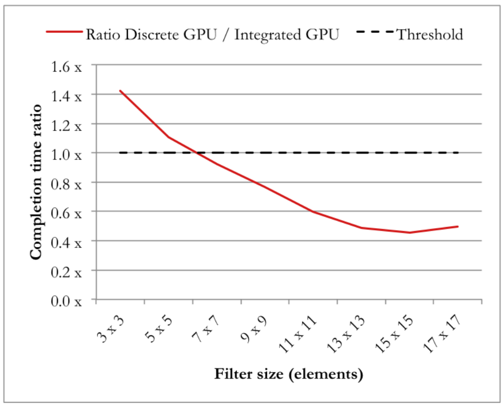

Figure 4 Convolution completion time ratio varying filter size: discrete / integrated GPU.

3.2 Computation Density

For each algorithm, we calculate the ratio between the number of operations performed on data and the amount of data transferred between the CPU and the GPU. We call this metric *Computation Density*(*CD*). The aim of this metric is to classify the efficiency of integrated versus discrete GPUs in executing a particular algorithm. As shown in table 2, saxpy and reduction have a constant CD while matrix multiplication is characterized by a CD that is linear on the data size. In matrix convolution, the CD depends both on the input matrix size and on the filter size. When the matrix size is much bigger than the filter area ($N \gg M^2$), the CD can be approximated to the following, which is constant on the input matrix size.

$$CD_{smallfilter} = 2M^2 \tag{1}$$

When the matrix size and the filter area are similar ($N \approxeq M^2$), the ratio can be instead approximated using the below formula, which is linear on the input size.

$$CD_{bigfilter} = \frac{2N^3}{2N^2} = N \tag{2}$$

The values of CD calculated for the algorithms taken into account suggest that the efficiency of integrated GPUs versus discrete GPUs is highly correlated to the balance between the amount of operations performed on data and the amount of data transferred to and from the GPU. In particular, when the CD is constant and below a certain threshold, the integrated GPU is the faster device. Whereas the CD is more than constant on the input size (e.g. linear, for matrix multiplication), the discrete GPU is instead capable of outclassing the integrated GPU. The most interesting test revealing this correlation is the matrix convolution, where CD can be considered constant on the input size except for matrixes and filters of similar sizes. Since the filter size is always smaller than the input matrix size, there are two chances for input matrix and filter to have similar sizes, that is decreasing the input matrix size and increasing the filter size. These two situations are the one discovered executing the algorithm on the testing platform (fig. 3 and 4).

Table 2 Computation Density of the tested algorithms

Saxpy	$\frac{N^2}{3N^2} = \frac{1}{3}$ (N^2 matrix size)
Reduction	$\frac{N}{N+1} \approx 1$ (N vector size)
Matrix multiplication	$\frac{2N^3}{3N^2} = \frac{2}{3}N$ (N^2 matrix size)
Matrix convolution	$\frac{2M^2N^2}{2N^2+M^2}$ (N^2 matrix size, M^2 filter size)

4 Conclusion

In this paper we shown that APU's GPUs and discrete GPUs can effectively accelerate different kind of computations, giving us the chance to specialize algorithms in order to obatain the best performances from an hybrid multi-GPU system. We also tried to correlate the test results with the characteristics of each specific algorithm, leading to a metric called *Computation Density*. Actually, many aspects can influence the completion time, such as thread synchronization, memory access patterns (influencing bank conflicts and coalescing) and loop unrolling. We are working to refine the metric to take into account all these aspects while trying to keep it as simple as possible. The following step is to take into account the CPU, which is particularly challenging due to it's different execution model and to the CPU-GPU memory sharing. In particular, memory sharing poses contention problems, since empoying the CPU to perform part of a computation may limit the memory access bandwidth and therefore the performances of the integrated GPU. Finally, we plan to use the Computation Density[1] to train a classifier on a large set of widely used computational patterns. The target is to employ machine-learning and parallel patterns discovery to partition and classify general purpose algorithms on the basis of the set of parallel computational patterns recognized during code analysis.

References

1 M. Daga, A. M. Aji, and W.-c. Feng. On the efficacy of a fused cpu+gpu processor (or apu) for parallel computing. In *Proceedings of the 2011 Symposium on Application Accelerators in High-Performance Computing*, SAAHPC '11, pages 141–149, Washington, DC, USA, 2011. IEEE Computer Society.

2 A. D. Malony, S. Biersdorff, S. Shende, H. Jagode, S. Tomov, G. Juckeland, R. Dietrich, D. Poole, and C. Lamb. Parallel performance measurement of heterogeneous parallel sys-

[1] Refined and eventually intergated with other metrics

tems with gpus. In *Proceedings of the 2011 International Conference on Parallel Processing*, ICPP '11, pages 176–185, Washington, DC, USA, 2011. IEEE Computer Society.

3 S. Ryoo, C. I. Rodrigues, S. S. Stone, J. A. Stratton, S.-Z. Ueng, S. S. Baghsorkhi, and W.-m. W. Hwu. Program optimization carving for gpu computing. *J. Parallel Distrib. Comput.*, 68:1389–1401, October 2008.

4 R. Vuduc, A. Chandramowlishwaran, J. Choi, M. Guney, and A. Shringarpure. On the limits of gpu acceleration. In *Proceedings of the 2nd USENIX conference on Hot topics in parallelism*, HotPar'10, pages 13–13, Berkeley, CA, USA, 2010. USENIX Association.

5 The Tradeoffs of Fused Memory Hierarchies in Heterogeneous Computing Architectures. K. Spafford, J. S. Meredith, S. Lee, D. Li, P. C. Roth and J. S. Vetter. *Future Technologies Group, Computer Science and Mathematics Division, Oak Ridge National Laboratory, 1 Bethel Valley Road-MS6173, Oak Ridge, TN 37831.*

6 G. de Bailliencourt. M-JPEG Decoding Using OpenCL on Fusion *AMD Fusion Developer Summit*, 2011.

7 Accelerating Lossless Data Compression with GPUs. R.L. Cloud, M.L. Curry, H.L. Ward, A. Skjellum and P. Bangalore. *July, 2011.*

8 S. Keely. Heterogeneous Kd-tree Construction on an APU *AMD Fusion Developer Summit*, 2012.

9 S. Venkatasubramanian, R. W. Vuduc. Tuned and Wildly Asynchronous Stencil Kernels for Hybrid CPU/GPU Systems. *Georgia Institute of Technology, College of Computing, School of Computer Science, 266 Ferst Drive, Altanta, Georgia, USA.*

10 D. Grewe and M.F.P. O'Boyle A Static Task Partitioning Approach for Heterogeneous Systems Using OpenCL. *School of Informatics, The University of Edinburgh, UK*, 2011.

11 C. Luk [1], and S.H.H. Kim [2]. Exploiting Parallelism on Heterogeneous Multiprocessors with Adaptive Mapping. [1] *SSG Software Pathfinding and Innovation, Intel Corporation, Hudson, MA. School of Computer Science.* [2] *Georgia Institute of Technology, Atlanta, GA.*

12 Multi-GPU and Multi-CPU Parallelization for Interactive Physics Simulations. E. Hermann[1], B. Ran[1], F. Faure[2],, T. Gautier[1] and J. Allard[1]. [1]*INRIA.* [2]*Grenoble University.*

13 M. D. Linderman, J. D. Collins, H. Wang and T. H. Meng. Merge: A Programming Model for Heterogeneous Multi-core Systems Abstract. *2011.*

14 V.J. Jimenez[1], L. Vilanova[2], I. Gelado[2], M. Gil[2], G. Fursin[3] and N. Navarro[2]. Predictive Runtime Code Scheduling for Heterogeneous Architectures. [1]*Barcelona Supercomputing Center (BSC).* [2]*Departament d'Arquitectura de Computadors (UPC).* [3]*ALCHEMY Group, INRIA Futurs and LRI, Paris-Sud University.*

15 Advanced Micro Device. *AMD Accelerated Parallel Processing with OpenCL. Revision 2.2. June, 2012.*

16 M. Miron Bernal, H. Coyote Estrada, J. Figueroa Nazuno. Code Similarity on High Level Programs. *Proceedings of the 18th Autumn Meeting on Communications, Computers, Electronics and Industrial Exposition. (IEEE - ROCC07). Acapulco, Guerrero, Mexico. 2007.*

17 N. Wu, S. M. M. Tahaghoghi. Evolving similarity functions for code plagiarism detection. *Honours Thesis. School of Computer Science and Information Technology. RMIT University. Melbourne, Australia. October, 2007.*

18 J. Dong[1], Y. Sun[2], Y. Zhao[1]. Design Pattern Detection by Template Matching. [1]*Computer Science Department, University of Texas, TX 75083, USA.* [2]*American Airlines, 4333 Amon Carter Blvd, Fort Worth, TX 76155, USA.*

19 J. Holewinski. PTX Back-End: GPU Programming with LLVM. *The Ohio State University. LLVM Developer's Meeting. November 18, 2011.*