Automatic Kernel Mapping for Functionally Heterogeneous Parallel Architectures

Robert Lyerly, Alastair Murray and Binoy Ravindran
Bradley Department of Electrical and Computer Engineering
Virginia Tech
Blacksburg, VA 24061
{rlyerly, alastair, binoy}@vt.edu

***Abstract*—Recently parallel architectures have entered every area of computing, from multi-core processors to massively parallel devices such as GPGPUs. These architectures have very different performance properties and capabilities. Through experiments with the "13 Dwarfs" and the NAS parallel benchmarks we show that different kernels map more efficiently to different architectures. We use these to motivate this position paper and we describe our proposal for an automated compiler-based tool to map kernels to the appropriate architecture. Our tool will allow the programmer, or auto-parallelizer, to describe parallelism using a mature model: OpenMP. This abstracts the underlying hardware-specific parallelism model, allowing the programmer to focus on describing the program, rather than non-portable optimizations. Our proposed tool will support an arbitrary number of architectures, but initially we are focusing on x86, Tilera and GPGPUs. The tool will be able to map code and transfer data to each of these architectures automatically and insert all necessary communication code. The mapping process will be based on predictive performance models that balance computer performance against data-transfer time.**

I. Introduction

In 2006 Asanovic et. al. [1] described "13 dwarfs" that represent key-problems in parallel computation. They claimed that these are the 13 key areas that parallel computation must support to allow performance scaling and included some that are not inherently parallel.

We use a subset of these key problems to investigate the characteristics of three parallel architectures: a Tilera TILE-Gx36 (36 VLIW cores at 1GHz), an AMD Opteron 6376 (16 superscalar cores at 2.3Ghz) and a NVIDIA GeForce GTX 560 Ti (448 cores at 1.5GHz). To have benchmarks which can run on all three architectures we started with a set of OpenCL implementations of algorithms that map to the "13 dwarfs" [6], six of which also have an OpenMP version available [3]. The remainder we ported to OpenMP manually.

These benchmarks cover 10 of the 13 dwarfs – the OpenCL benchmark suite only includes 11 and we dropped the finite state machine dwarf as it is inherently serial (and we are mostly interested in parallel architectures). Regardless, the 10 dwarfs we show highlight a number of different kernel characteristics.

Figure 1 shows the performance of these architectures on each kernel. It can be seen that five map most efficiently to x86, and five to the GPU, thus demonstrating that different computational problems can be computed most efficiently on different hardware.

Tilera can be seen to consistently under-perform in comparison to the other two architectures, but it is actually far lower power. We do not present power numbers in this paper as we do not yet have the infrastructure to measure this, but the reference sheet "typical power" of the entire Tilera board is about 50% that of the x86 CPU alone, and about 25% that of the GPU board. This means that as long as it runs in no more than half the time of the x86 (e.g. astar in figure 1) it will use less power to complete the task. This is especially important when power efficiency is critical (such as data-center, HPC or embedded systems contexts). The Tilera, however, can also be more power hungry than an x86 processor if it takes too long to complete a task (e.g. gem). So, as with performance, different kernels can be mapped to different architectures to optimize power usage.

Additionally, to ensure that these trends translate from kernels to applications we also considered the NAS Parallel Benchmarks [2]. We took the official OpenMP implementation (on x86 and Tilera) and an OpenCL implementation [17] for the GPU. The results of running with dataset B, shown in figure 2, show that five are fastest on the x86, and three are fastest on the GPU.

This motivates our belief that applications can see increased performance if partitioned across multiple heterogeneous parallel architectures, picking the ideal architecture per-kernel.

II. Mapping Tool Design

To target this heterogeneous partitioning problem we propose a compiler-based "source-to-source" tool. This will take an OpenMP program as input and produce a partitioned "program" as output. The partitioned program will in-fact be two or more tightly coupled programs targeted to different hardware. This partitioning will follow an accelerator model where the program runs

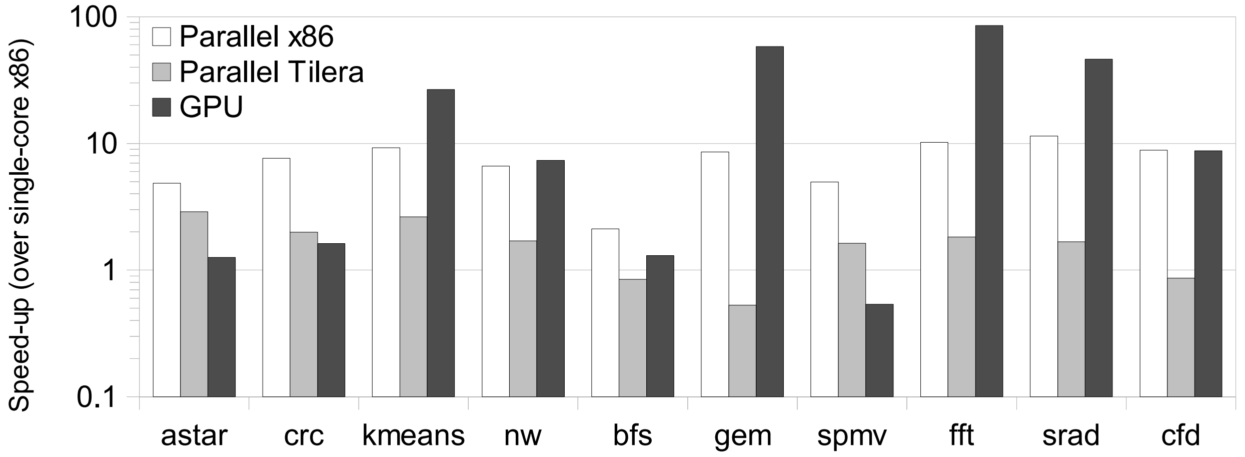


Fig. 1. The performance of each "dwarf" on different architectures, relative to a single-core x86 processor. These represent a collection of GPU benchmarks, the GPU runs the original OpenCL code, the x86 and the Tilera run OpenMP versions. In all cases program start-up time is excluded, so the time to compile the OpenCL kernels or create the OpenMP thread pool is not included.

on a host architecture, which can send work to other architectures, as shown in figure 3.

At a high-level, the tool characterizes each function to decide which architecture it would most closely map to. Primarily it will focus on functions containing parallel loops (marked by OpenMP pragmas), but these do not necessarily have to be leaf nodes in the call graph – partitioning an entire sub-graph is also considered. This characterization will be used to decide how to partition the call-graph in conjunction with expected data-transfer requirements. Once the partition boundaries have been decided the partitioned code will be separated out and run-time code will be added to transfer control.

This run-time stub will have final control over where to run each partition. As the exact data-transfer requirements can only be known at run-time, and may vary with program input, the stub can decide to not send the work to an accelerator but to run it on the host. Additionally, if the ideal accelerator is currently busy the stub can decide to either wait or send the work elsewhere, thus allowing the program to adapt to external workloads.

We have begun to develop this tool primarily within the ROSE compiler frame-work [11], initially targeting x86, Tilera and GPGPUs. We also use a GCC plugin to extract the "features" of each function and the WEKA [10] machine learning software to prototype performance predictors – together these perform the characterization of each function. Finally, x86 is the host architecture, but to run code on the Tilera platform we use MPI to transfer data and control and we use existing tools to translate code to CUDA to run on GPGPUs.

A. Characterization: Prediction

To decide how to partition a program it is necessary to predict which architecture each function is best suited to. We propose utilizing machine learning to produce this prediction. We chose this direction as machine learning has been shown to be capable of finding the structure underlying different models. This is essential for providing portable performance as the interactions between code, the compiler and the hardware architecture are extremely complicated to manually model. We are currently examining two possible machine learning models: an execution time predictor and a classifier. The predictor should be able to predict how long a function will take to execute on each architecture, whereas the classifier will pick the architecture that a function is most-suited for. We are currently prototyping these in WEKA based on linear models but plan to investigate more modern machine learning methods. Each of these models require very different program descriptions to operate. The execution time predictor requires low-level instruction counts supported by profiling data, as execution time varies with inputs. A classifier, however, does not require profiling data and can produce an answer from higher-level features of a function but without a predicted execution time it is difficult to balance computation with data transfer requirements. Both of these approaches involve an off-line training stage, but only the final model is integrated into the compiler so the effect on compilation time is negligible.

B. Characterization: Feature Extraction

To gather the features of the application required to support the two predictors described above we have implemented a GCC plugin. To try to describe a function this captures the run-time functionality, such as integer/floating point calculations, logic operations, memory accesses, control-flow statements, etc. It examines GCC's internal intermediate representation (GIM-

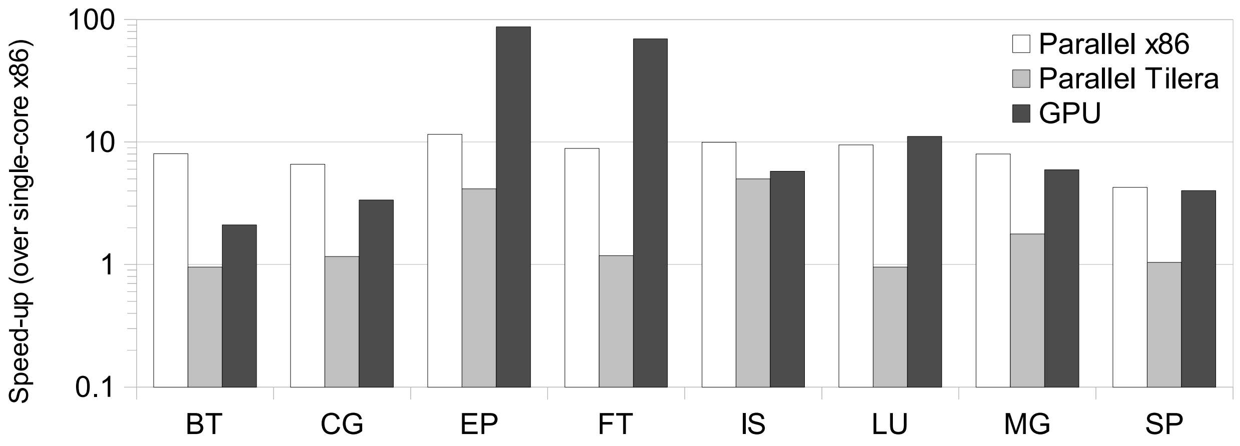


Fig. 2. The performance of a subset of the NASA Parallel Benchmarks on different architectures, relative to a single-core x86 processor. The x86 and Tilera run OpenMP code, the GPU runs hand-tuned OpenCL code. For these benchmarks the complete program run-time is reported, including the time to build the OpenCL kernels.

PLE) to collect this data, and produces a human-readable/machine-processable description of each function.

C. Mapping Kernels: Processing

We use an annotated version of the input program's call-graph to decide the mapping of kernels to partitions. We annotate the call-graph such that the edge between two nodes (i.e. two functions) has a weight equal to the amount of data that must be transferred to and from the function. This may not be known at compile-time, so run-time checks can be inserted. Additionally, each node in the graph is annotated with it's predicted performance on each compatible architecture. We consider a computational kernel to be a sub-graph of the call-graph with a single entry point, e_k, that marks the "root" of the graph. Generally, if a sub-graph represents a kernel and has multiple exit points (i.e. it calls a function outside of its partition) then this can be resolved by cloning the external function. Similarly even if a sub-graph has multiple entry points these are just another partition's exit points, so again, these can be cloned. For example, a frequently used helper function may be connected to many partitions, but assuming it is itself a leaf function then it can cloned without issue.

To decide on which partition to place a kernel, a traversal beginning at e_k is performed. For each architecture the traversal accumulates the predicted run-times of every node in the sub-graph producing a per-architecture prediction for the total run-time of the kernel. Finally, the edge weight (the amount of data that must be transferred to and from e_k) is considered for each partition; for x86, this cost is zero, but for the Tilera and GPU, the time to transfer the data over PCIe must be included in the mapping. If the exact data-transfer requirements are known at compile time then kernel can be mapped directly to the architecture with the minimal execution-time. In general, however, data-transfer requirements can only be known at run-time so code can be produced for both the host and the optimal accelerator so that the mapping decision can be deferred to run-time. This also now allows the program to adapt to external workload meaning it can run on a sub-optimal architecture if that means it will complete more quickly than waiting for the ideal hardware.

D. Mapping Kernels: Strategies

There are many strategies that can be used to decide the mapping of kernels to devices, based on the characteristics of the application and the needs of the user. The most obvious mapping strategy is mapping kernels to accelerators to obtain the maximum possible speedup. In this instance, the tool will seek to minimize the predicted run-times and data transfer overheads of the kernel subgraph – the kernel will be mapped to the architecture with the shortest run-time. A variation of this idea involves using the predicted run-times to distribute a percentage of particularly large kernels across multiple devices, increasing the parallelism at a very coarse-grained level. Another strategy the tool will consider is optimizing for power usage. In some settings where power usage is a limiting factor (e.g. server environments), utilizing machines efficiently is a key point. In this situation, the Tilera coprocessor becomes an attractive option; although it does not match the computational power of modern x86 processors or a GPGPU, it consumes less than 50% of the power of an AMD Opteron 6376 and less than 25% of the power of an NVIDIA GTX 560; the Tilera is a strong candidate in this space. Additionally, the partitioner can

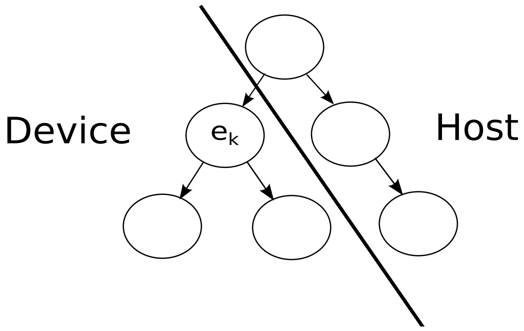
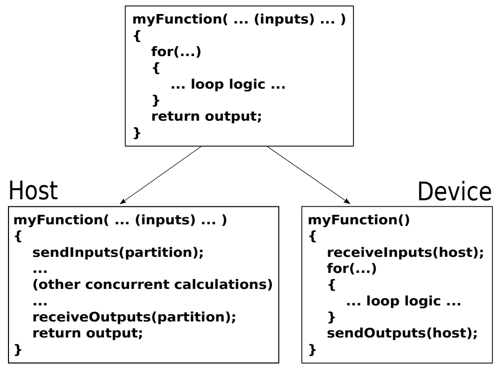


Fig. 3. The call-graph for an application. The kernel, represented by the sub-tree rooted at entry function e_k, is analyzed and mapped to a device based on the machine learning model. The kernel does not have to be a tree, but it must have a single entry/exit point, though function cloning can help satisfy this requirement in some cases.



```
myFunction( ... (inputs) ... )
{
   for(...)
   {
      ... loop logic ...
   }
   return output;
}
```

```
myFunction( ... (inputs) ... )
{
   sendInputs(partition);
   ...
   (other concurrent calculations)
   ...
   receiveOutputs(partition);
   return output;
}
```

```
myFunction()
{
   receiveInputs(host);
   for(...)
   {
      ... loop logic ...
   }
   sendOutputs(host);
}
```

Fig. 4. Code transformation performed by the partitioner. The original function is transformed into a stub for communication while the kernel is moved to the specified partition.

map kernels speculatively, if the user expects external workloads to impact the performance of the application then the kernel can run on one of several candidate architectures. In this instance, the tool can speculatively map a kernel to these different architectures at compile time so that if the "best" device for the kernel is busy with another application, the kernel can be seamlessly run on another architecture. Supporting this will require run-time decision making based on information obtained from the operating system about system load and device availability.

E. Code Refactoring

Once a kernel has been mapped to an architecture, the partitioning tool must perform the code refactoring necessary to move the kernel to another platform, including all necessary handshaking and data transfer.

Creating Partitions: The partitioner either must be supplied with a configuration file or it must query the system to determine which platforms are available for the partitioner to utilize. If there are functions that are to be run on a particular platform, it creates an empty partition for it.

Analysis of Functions: The partitioner must determine for each kernel whether it is suitable for partitioning. Function declarations without a visible body (e.g. library functions) cannot be partitioned for several reasons: first, the code to perform communication cannot be inserted and second, parallelization analysis cannot be performed. During analysis, we must discover the inputs and outputs of a kernel. This becomes especially important with separate memory spaces; any change in the device memory space must be reflected in the host memory space. Function argument lists and return statements provide the basic inputs and outputs of a function, but the semantics of global variables and function side-effects must be obeyed. For those functions that use dynamically allocated data, data transfer sizes must be discovered at run-time – the partitioner inserts the necessary probes to resolve these. During analysis, the tool also maintains a list of all functions used in the kernel subgraph. Any functions called during kernel execution are also analyzed to make sure they can be placed on a separate partition, with some relaxed constraints. When the program is restructured, they are moved with the kernel to the other partition.

Re-structuring the Program: Once a function has been analyzed and deemed appropriate for partitioning, it must be moved to the other partition. We must, however, also insert code to perform the handshaking necessary to launch the kernel on the other partition – figure 4 shows how the code is transformed. The function body of the kernel is lifted out of the abstract syntax tree (AST) of the host partition and moved to the device partition's AST. The original function body on the host is transformed into a simple stub which encapsulates all communication with the device. First, a call on the host side will notify the device that it wants to execute a particular kernel. Then, calls are inserted to transfer inputs to the kernel; similarly, calls are inserted to transfer outputs back to the host partition. In this way, all communication between the host and accelerators is handled by the partitioner, relieving the programmer of hand-coding data movement. To know how much data to transfer, the size of every memory block must be known. To support this we have written a memory management library that wraps `malloc` and siblings, and provides hooks for the partitioner to record the sizes of static memory allocations. Using this library, the run-time mapper can request the size of the memory block associated with any pointer. The calls to transfer data are specific to the accelerator. For the Tilera board, MPI calls are used to communicate with the processor over PCIe. The GPU requires calls to the CUDA run-time (`cudaMalloc` and `cudaMemcpy`) to transfer data. If the kernel remains on the host CPU, no data movement (or, code refactoring) is necessary.

Final Steps: Once the program has been restructured into partitions, each partition must be compiled for its particular device. In the case of x86 and Tilera, this involves straightforward building with a compiler that supports OpenMP/MPI. For the GPU partition the kernels must be converted to a GPU language such as CUDA. Much work has been put into investigating methods of performing this transformation, and one of these tools [13, 15] will be used to refactor the GPU partition into CUDA code.

III. Related Work

Characterizing programs is a common step in traditional manual optimization, but more recently there has been research into doing this automatically. Milepost GCC by Fursin et. al. [7] uses machine learning to choose optimal optimization options for an application. They provide an in-depth discussion of how program features describe an application but they focus on features and models that attempt to choose the best subset of optimization flags, rather than features that characterize a program's performance. Thoman et. al. [18] describe ways to characterize platforms based on features pertinent to performance on heterogeneous systems, such as arithmetic throughput, complexity of the memory system, branching penalties and run-time overheads. They developed a micro-benchmark suite, μCLbench, that contains a set of small OpenCL benchmarks that analyze these characteristics. They used this information, however, to hand optimize applications rather than attempting to build a tool to perform automatic optimization.

Several groups have studied partitioning and mapping of programs on heterogeneous systems; in particular, there has been a focus on systems containing a multi-core CPU and a GPU. Kim et. al. [12] proposed SnuCL, a run-time system targeting heterogeneous clusters. SnuCL expands on OpenCL semantics so that a cluster is simply viewed as a collection of compute devices (each multi-core CPU or GPU is considered a computation device). While the SnuCL run-time maps kernels to devices, it does not make any distinction between CPU and GPU devices, ignoring the opportunity to select the most appropriate architecture for a kernel. Luk et. al. [14] describe an adaptive approach, used in their heterogeneous programming framework Qilin, for executing a kernel simultaneously on a CPU and GPU. Their approach uses curve fitting to build an empirical model of the application's execution time that can be analyzed to find a near-optimal mapping of the kernel onto the two platforms. Their approach, however, requires a training run for each new application to construct the initial model, the tool cannot learn from previous data. Grewe and O'Boyle [8] present a methodology for statically mapping OpenCL programs to run a single computational kernel simultaneously on a CPU and GPU. They use a machine learning-based approach to construct prediction models based on static program features. Features are collected at compile time for a given application and the machine learning model makes a prediction for mapping the OpenCL kernels, but no partitioning is performed so the OpenCL kernels must be provided by the programmer. They did, however, later extend this [9] with a framework that generates OpenCL kernels from OpenMP parallel blocks and decides on which platform (CPU vs. GPU) the computational kernel should run. Their framework applies several GPU-specific optimizations to the generated OpenCL code and then uses a machine learning model to decide at run-time on which architecture to run. They only consider CPU + GPU, and not the effect of external workload.

Other works have also looked at compiling non-GPU code for GPUs. Lee et. al. [13] developed OpenMPC, a tool built on top of Cetus [4] that attempts to translate the task-level parallelism of OpenMP into the data-level parallelism of CUDA. Par4All [15] expands this scope by accepting sequential C and Fortran as input and generating CUDA, OpenCL, OpenMP or MPI as output. Neither of these, however, consider partitioning.

Finally, other researchers have considered external workloads in non-partitioning contexts. Emani et. al. [5] applied machine learning to develop a heuristic that determines at run-time how many threads a given OpenMP parallel block should use based on the external workload of the system. The heuristic combines static program features (generated at compile-time) and external workload information (obtained via calls to a run-time library) to make the decision. Alternatively, Saez et. al. [16] support external workloads for single-ISA heterogeneous systems via the operating-system scheduler, but do not support fully heterogeneous systems.

IV. Conclusion

We have presented our position that parallel applications need to be partitioned across heterogeneous hardware to achieve optimal performance and have shown the differing performance characteristics of three separate architectures to motivate this. We have proposed the design of a tool that can perform this partitioning, considering either performance or power while also being capable of adapting to external workload.

Acknowledgements

This work is supported by the US Office of Naval Research under Contract N00014-12-1-0880.

References

[1] Krste Asanovic, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Kurt Keutzer Husbands, David A. Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams, and Katherine A. Yelick. The landscape of parallel computing research: A view from Berkeley. Technical report, Electrical Engineering and Computer Sciences, University of California at Berkeley, 2006.

[2] D.H. Bailey, E. Barszcz, J.T. Barton, D.S. Browning, R.L. Carter, L. Dagum, R.A. Fatoohi, P.O. Frederickson, T.A. Lasinski, R.S. Schreiber, H.D. Simon, V. Venkatakrishnan, and S.K. Weeratunga. The NAS parallel benchmarks. *International Journal of High Performance Computing Applications*, 5(3):63–73, September 1991.

[3] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang-Ha Lee, and Kevin Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *IEEE International Symposium on Workload Characterization (IISWC '09)*, pages 44–54, Austin, TX, USA, October 2009.

[4] Chirag Dave, Hansang Bae, Seung-Jai Min, Seyong Lee, Rudolf Eigenmann, and Samuel Midkiff. Cetus: A source-to-source compiler infrastructure for multicores. *Computer*, 42(12):36–42, December 2009.

[5] Krishna Murali Emani, Zheng Wang, and Michael F.P. O'Boyle. Smart, adaptive mapping of parallelism in the presence of external workload. In *2013 International Symposium on Code Generation and Optimization (CGO '13)*, Shenzhen, China, February 2013.

[6] Wu-chun Feng, Heshan Lin, Thomas Scogland, and Jing Zhang. OpenCL and the 13 dwarfs: A work in progress. In *Proceedings of the third joint WOSP/SIPEW international conference on Performance Engineering (ICPE '12)*, pages 291–294, Boston, MA, USA, 2012.

[7] Grigori Fursin, Yuriy Kashnikov, Abdul Wahid Memon, Zbigniew Chamski, Olivier Temam, Mircea Namolaru, Elad Yom-Tov, Bilha Mendelson, Ayal Zaks, Eric Courtois, Francois Bodin, Phil Barnard, Elton Ashton, Edwin Bonilla, John Thomson, Christopher K.I. Williams, and Michael O'Boyle. Milepost GCC: Machine learning enabled self-tuning compiler. *International Journal of Parallel Programming*, 39:296–327, 2011.

[8] Dominik Grewe and Michael F. P. O'Boyle. A static task partitioning approach for heterogeneous systems using OpenCL. In *Proceedings of the 20th international conference on Compiler Construction (CC '11)*, pages 286–305, Saarbrücken, Germany, 2011.

[9] Dominik Grewe, Zheng Wang, and Michael F.P. O'Boyle. Portable mapping of data parallel programs to OpenCL for heterogeneous systems. In *2013 International Symposium on Code Generation and Optimization (CGO '13)*, Shenzhen, China, February 2013.

[10] Mark Hall, Eibe Frank, Geoffrey Holmes, Bernhard Pfahringer, Peter Reutemann, and Ian H. Witten. The WEKA data mining software: An update. *ACM SIGKDD Explorations Newsletter*, 11(1):10, November 2009.

[11] ROSE Compiler Infrastructure. http://rosecompiler.org/, March 2013.

[12] Jungwon Kim, Sangmin Seo, Jun Lee, Jeongho Nah, Gangwon Jo, and Jaejin Lee. SnuCL: an OpenCL framework for heterogeneous CPU/GPU clusters. In *Proceedings of the 26th ACM international conference on Supercomputing (ICS '12)*, pages 341–352, Venice, Italy, 2012.

[13] Seyong Lee and Rudolf Eigenmann. OpenMPC: Extended OpenMP programming and tuning for GPUs. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC '10)*, pages 1–11, Washington, DC, USA, 2010.

[14] Chi-Keung Luk, Sunpyo Hong, and Hyesoon Kim. Qilin: exploiting parallelism on heterogeneous multiprocessors with adaptive mapping. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-42)*, pages 45–55, New York, NY, USA, 2009.

[15] Par4All Project. http://www.par4all.org/, March 2013.

[16] Juan Carlos Saez, Daniel Shelepov, Alexandra Fedorova, and Manuel Prieto. Leveraging workload diversity through OS scheduling to maximize performance on single-ISA heterogeneous multicore systems. *Journal of Parallel and Distributed Computing*, 71(1):114–131, 2011.

[17] Sangmin Seo, Gangwon Jo, and Jaejin Lee. Performance characterization of the NAS parallel benchmarks in OpenCL. In *Proceedings of the 2011 IEEE International Symposium on Workload Characterization (IISWC '11)*, pages 137–148, Austin, TX, USA, November 2011.

[18] Peter Thoman, Klaus Kofler, Heiko Studt, John Thomson, and Thomas Fahringer. Automatic OpenCL device characterization: guiding optimized kernel design. In *Proceedings of the 17th international conference on Parallel processing (Euro-Par '11)*, pages 438–452, Bordeaux, France, 2011.