Enabling Portable Optimizations of Data Placement on GPU

A new software framework, called Porple, makes it possible to automatically enhance data placement across GPUs. Through Porple, a GPU program's data is placed appropriately on memory on the fly, customized to the input dataset. When new memory systems arrive, Porple adapts the placements accordingly. Experiments on three types of GPU systems show that Porple consistently finds optimal or near-optimal placement.

Guoyang Chen
North Carolina State University

Bo Wu
Colorado School of Mines

Dong Li
University of California, Merced

Xipeng Shen
North Carolina State University

For a massively parallel architecture such as a GPU, it is important to narrow the gap between the memory throughput that its hundreds of cores demand and the amount that memory systems can provide. This issue has prompted the adoption of various types of memory and increasingly complex designs in modern memory systems. For example, an Nvidia Kepler GPU has more than eight types of memory (global, texture, shared, constant, various cache, and so on), with some on chip, some off chip, some directly manageable by software, and some not. The various memory types each have their own size, properties, and access constraints. Having them in a single system helps with throughput and flexibility, but it also imposes challenges for their effective usage.

Studies have shown that finding the suitable kinds (or pieces) of memory to place data—called the *data placement problem*—is essential for program performance. Different placements of data on memory systems often cause substantial differences in program performance,[1] which our experiments echoed. The importance of data placement is especially prominent on memory-bound programs: as Figure 1 shows, better data placements brings 1.31 to 2.63 times speedup on the six CUDA kernels over the programmer-determined data placements.

Most existing GPU programming models (such as OpenACC and OpenCL) require programmers to indicate the appropriate data placements for a program—a task that has proven to be daunting in practice. Because of the complexity and continuous evolvement of memory, it is often difficult for programmers to tell which data placements fit a program. For some programs, the suitable placements differ across their inputs, making placement even more difficult.

Prior solutions fall into two categories: offline autotuning, and a rule-based approach. The first tries many different placements and measures the performance on some training runs.[2] The approach is time consuming and cannot easily adapt to the changes in program inputs or memory systems. The second approach uses some high-level rules derived empirically from many executions.[1] These high-level rules are straightforward, but they often fail to produce suitable placements, and

Published by the IEEE Computer Society

their effectiveness degrades further when memory systems evolve across generations.

In response to these challenges, we developed Porple, a portable data placement engine that enables a new way to solve the data placement problem.

Our solution: Porple

As Figure 2a shows, Porple contains three key components:

- a memory specification language (MSL) for architects to provide memory specifications,
- a compiler called *Porple-C* for revealing the program's data access patterns and staging the code of the program of interest for runtime adaptation, and
- an online data placement engine called *Placer* that consumes the specifications and access patterns to find the best data placements at runtime.

Porple has three desirable properties that previous solutions lack: extensibility, cross-input adaptivity, and generality.

Extensibility

Memory systems often manifest some changes in a new design generation. For a solution to have lasting value, it must be easy to extend to cover a new memory system. Our solution features MSL, a carefully designed small specification language. Figure 2b shows the excerpt of the syntax of MSL. MSL provides a simple, uniform way to specify a type of memory and its relations with other pieces of memory in a system. Each piece of memory in the system has one or more entries in the MSL specification, indicating its size, latency, number of banks, relations with other memory entries (for example, the memory hierarchy), and access constraints (such as read only or 2D data locality). It also allows the specification of the overlapping effects of multiple memory transactions (through a field called *concurrencyFactor*).

A feature of MSL worth noting is its treatment of special properties of memory. GPU memory has special properties. For instance, on Tesla K20, for shared memory, when two accesses to the same bank happen, they must

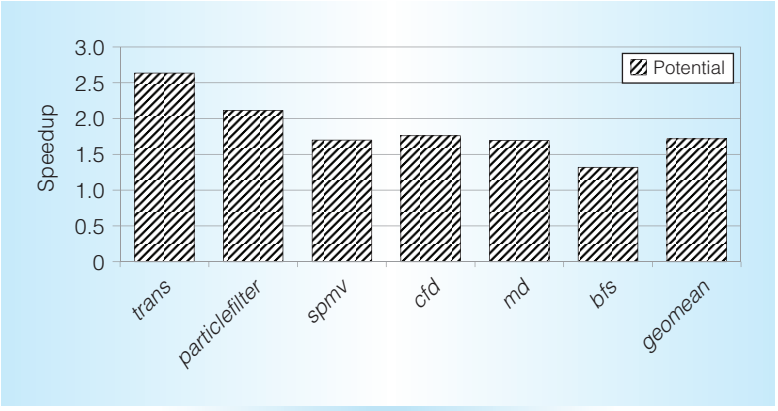
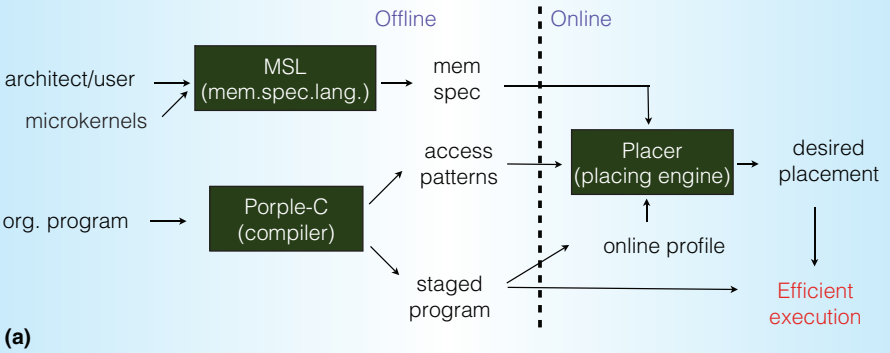


Figure 1. Better data placements yield significant speedups on Tesla K20. The bars show the speedups of the programs when the best data placements, obtained through exhaustive search, are applied to the programs. The baseline version uses the default data placements determined by programmers.

be served serially. For global memory, two accesses by the same warp can be coalesced into one memory transaction if their target memory addresses belong to the same segment. For texture memory, accesses can benefit from 2D locality. For constant memory, the accesses to the same address are satisfied instantly by value broadcasting, but when addresses differ, they must be served one after one (see http://docs.nvidia.com/cuda/cuda-c-programming-guide/#axzz3Ou3lVXqF).

How to allow a simple expression of all these various properties is a challenge for MSL design. We address this question based on an insight that all these special constraints are essentially about the conditions for multiple concurrent accesses to a memory to get serialized. Accordingly, MSL introduces a field called *serialCondition* that lets specification writers use simple logical expressions to express all those special properties. For instance, the expression for shared memory, *block{word1!=word2 && word1%banks==word2%banks}*, claims that when the words accessed by two threads in the same thread block are different and fall onto the same bank (a *bank conflict*), the two accesses get serialized. This design makes it simple to express various special properties of GPU memory in a unified manner.

MSL offers an underlying vehicle for Porple to treat the complex GPU memory automatically. By encapsulating memory complexities



(a)

```
Syntax:
• specList ::= processorSpec memSpec*
• processorSpec ::= die=Integer tpc; tpc=Integer sm; sm=Integer core;
  end-of-line
• memSpec ::= name id swmng rw dim size blockSize banks latency upperLevels
      lowerLevels shareScope concurrencyFactor serialCondition ; end-of-line
• name ::= String
• id ::= Integer
• swmng ::= Y | N // software manageable or not
• rw ::= R|W|RW // allow read or write accesses
• dim ::= na | Integer // special for arrays of a particular dimensionality
• sz ::= Integer[K|M|G|T|ε][E|B] // E for data elements
. . . . . .
```

(b)

Figure 2. The Porple portable data placement engine. (a) A high-level overview of Porple's structure, including the memory specification language (MSL), Porple-C compiler, and Placer online data placement engine. (b) An excerpt from the syntax of the MSL in Backus-Naur form (BNF).

within a specification, the design simplifies the adaptation of data placements across memory systems: For a new memory system, as long as its MSL specification is given (for example, by an architect), the other two components of Porple can automatically infer and adjust the data placements of a program when it runs on this new system. Note that architects or experts need to write the specification in MSL only once per GPU model; it can then be used by all other programmers and users.

Cross-input adaptivity

The second desirable property is input adaptivity. Different input datasets of a program could differ in size and trigger different data access patterns, and thus demand different data placements. Because program inputs are not known until runtime, the data placement optimizer should be able to work on the fly, which entails two requirements.

Efficient data placement engine. The first requirement is to employ a highly efficient data placement engine that works with minimum runtime overhead. Our solution features a lightweight memory performance model for quickly assessing different data placement plans. The model takes as input the program data access patterns and architecture specifications (written in MSL) and outputs the given program's estimated memory performance using a certain data placement. Unlike prior GPU performance models that focus on accuracy,[3] our model strives to be easy to build and quick to use for meeting

online usage requirements. The model is path based. It estimates the amount of time taken by data transfers over each of the data transfer paths in the system. Nvidia Kepler GPUs, for instance, have three data transfer paths: one between a core and the global memory, one between a core and the texture memory, and one between a core and the constant memory. Note that the first also covers accesses between a core and the shared memory, because those transfers take a part of that first path. Similarly, the second also covers accesses to and from the read-only cache. Because the three paths transfer data concurrently, the performance model uses the maximum of the times estimated on those paths to assess the quality of a data placement.

Specifically, let P be the set of paths, $A(p)$ be the set of arrays whose accesses take path p, and N_{ij} be the number of memory transactions of array i that happen on memory whose ID equals j. Porple assesses the quality of a data placement plan through the following performance model:

$$\max_{p \in P} \left\{ \sum_{i \in A(p)} \sum_{j \in \text{memHier}(i)} N_{ij} T_j \alpha_j \right\}.$$

The inner summation estimates the total time that accesses to array i incur, and the outer summation sums across all arrays going through path p. In the formula, memHier(i) is the memory hierarchy that accesses to array i go through, and T_j is the latency of a memory transaction on memory j. The parameter j is the memory's concurrency factor, which takes into account that multiple memory transactions can be served concurrently. For instance, multiple memory transactions can be served concurrently on the global and texture memory (hence, for them, $0 < j < 1$), whereas only one memory transaction can be served on the constant memory ($j = 1$). The MSL specification gives the concurrency factor values; in our experiments, we use 0.2 for the global and texture memory and 1 for others.

Additional features of the runtime placement engine include an open design that allows easy adoption of fast algorithms for searching for the best placement, and an agile way to detect data access patterns. These features allow the placement engine to work at runtime to examine potential placement plans and identify the best one before a kernel gets invoked.

Staging code to be agnostic to placement. The second requirement for on-the-fly enhancement of data placement is the ability to transform the original program such that it can work with an arbitrary data placement decided by the data placement engine at runtime. A program written on a system with multiple types of memory typically does not meet the requirement because of its hard-coded data access statements, which are valid only under a particular data placement (for example, declaration *_shared_* for an array on GPU shared memory).

The Porple-C compiler transforms a program into a placement-agnostic form. The form is equipped with some guarding statements such that executions of the program can automatically select the appropriate version of code to access data according to the current data placement. A complexity with this solution is the tension between the size of the generated code and the overhead of the guarding statements, which we address with a combination of coarse- and fine-grained versioning.

Figure 3 illustrates this idea. The coarse-grained versioning creates multiple versions of the GPU kernel, each corresponding to one possible placement of the arrays (for example, in Figure 3b, kernel 1 is for the case when A0 is in texture and A1 is in shared memory). The appropriate version is invoked through a runtime selection based on the result from the Placer (see Figure 3a). When there are too many possible placements, the coarse-grained versions are created only for some of the placements (for example, the five most likely ones); for all other placements, a special copy of the kernel is invoked. This copy has fine-grained versioning on data accesses (see Figure 3c). The combination of the two levels of versioning helps strike a balance between code size and runtime overhead.

These techniques make it possible for the Placer to be invoked at execution, such that it can find the appropriate data placement for the current run of the program.

```
// host code
…
// coarse-grained version selection
// TXR: texture; SHR: shared mem
if (memSpace[A0_id]==TXR &&
memSpace[A1_id]==SHR)
       kernel_I(…); // Version 1
else
       kernel_others(…); // Version 2
```

(a)

```
// Version 1 code (kernel_I)
{…
       // code for statement: A1[j] = A0[i]
       sA1[...] = tex1Dfetch(A0tex, i);
…}
```

(b)

```
// Version 2 code (kernel_others)
{…
       // code for statement: A1[j] = A0[i]
       switch (memSpace[A0_id]){
         case GLB: _tempA0 = A0[i]; break;
         case GLR: _tempA0 = __ldg(&A0[i]); break;
         case SHR: _tempA0 = sA0[...]; break; // use local index
         case CST: _tempA0 = cBuffer[i]; break;
       } // GLB: global; GLR: read-only global; TXR: texture;
          // SHR: shared; CST: constant
       if (memSpace[A1_id]==SHR)
         sA1[...] = _tempA0; // use local index
       else
         A1[j] = _tempA0;
…. }
```

(c)

Figure 3. Illustration of placement-agnostic code. (a) Host code. (b) Implementation of $A1[j] = A0[i]$ in version 1. (c) Implementation of $A1[j] = A0[i]$ in version 2. The combination of coarse-grained kernel-level versioning and fine-grained statement-level versioning strikes a balance between code size and runtime overhead.

Generality

The third desirable property is good generality. Data placement is important for both regular and irregular programs (our results show an even higher potential on irregular programs). A good solution to the data placement problem should thus apply to both kinds of programs. Here, the main challenge is in finding out the data access patterns of irregular programs, because they typically are not amenable for compiler analysis. Our solution is to employ a hybrid method. In particular, the Porple-C compiler tries to figure out data access patterns through static code analysis. When the analysis fails, it derives a simplified function from the program with some key memory access patterns captured. The function contains some recording

Table 1. Description of the benchmarks we used to evaluate Porple.

Benchmark	Source	Description	Irregular
mm	CUDA software development kit (SDK)	Dense matrix multiplication	No
convolution	CUDA SDK	Signal filter	No
trans	CUDA SDK	Matrix transpose	No
reduction	Scalable Heterogeneous Computing (SHOC)	Reduction	No
fft	SHOC	Fast Fourier transform	No
scan	SHOC	Scan	No
sort	SHOC	Radix sort	No
traid	SHOC	Stream triad	No
kmeans	Rodinia	*k*-means clustering	No
particlefilter	Rodinia	Particle filter	Yes
cfd	Rodinia	Computational fluid	Yes
md	SHOC	Molecular dynamics	Yes
spmv	SHOC	Sparse matrix vector multi.	Yes
bfs	SHOC	Breadth-first search	Yes

instructions to characterize memory access patterns. At runtime, the function runs only for a short period of time, with strictly controlled overhead.[4] This hybrid method avoids runtime overhead when possible and at the same time makes the solution broadly applicable.

With Porple, an architect or expert user can easily specify the configurations of a memory system in MSL. With this specification, programmers no longer need to worry about data placement. For a given program, the Porple compiler and runtime can automatically equip an execution of the program with data placements suitable to the underlying memory system.

Evaluation

We have developed a prototype of Porple for Nvidia GPUs.

Methodology

We choose 14 benchmarks to evaluate Porple. Table 1 shows the details of the benchmarks. They include all level-1 benchmarks from the Scalable Heterogeneous Computing (SHOC) benchmark suite[5] and others from the Rodinia suite[6] and CUDA software development kit. To study Porple's portability, we use three different GPUs on different operating systems (see Table 2).

Table 2. The GPUs we used to study Porple's portability.

Name	GPU card	OS	CUDA
C1060	Tesla C1060	Linux-3.11	5.5
M2075	Tesla M2075	Linux-2.6	5.5
K20c	Nvidia K20c	Linux-3.13	6.5

The memory systems of the GPUs differ significantly. For instance, C1060 has no data cache for global memory accesses, M2075 uses both L1 and L2 caches for caching global loads, and K20c does not use the L1 cache to cache global loads. All reported timing results include all runtime overhead.

Results

We first examine the results on irregular benchmarks, which are the most difficult to handle because of their complex, irregular data access patterns. Figure 4a shows the speedups on K20c. The rule-based approach is the state of the art,[1] whereas the upper bound is what the best placement can give, obtained through exhaustive measurement. Porple achieves an average 1.69 times (up to 2.11 times) speedup, close to the upper bound (1.71 times speedup). The rule-based approach gets only 36 percent speedup because it cannot fully exploit the architecture

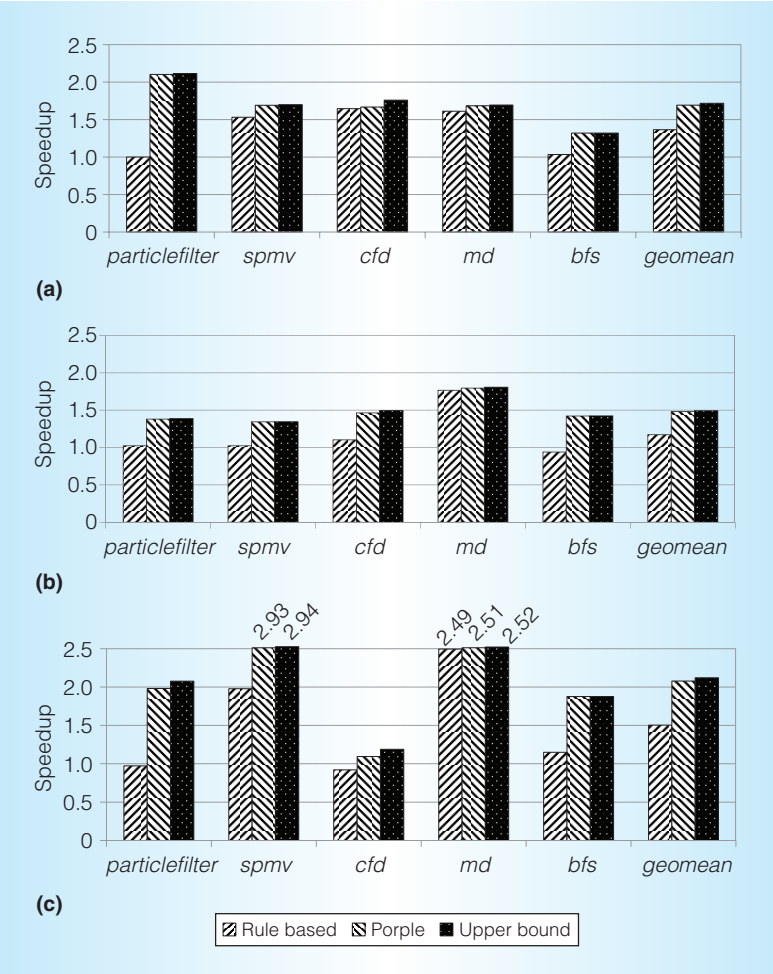


Figure 4. Speedup of irregular benchmarks on three generations of GPU. (a) Tesla K20. (b) Tesla M2075. (c) Tesla C1060. Porple outperforms the rule-based method on three GPUs and almost reaches the upper bound.

properties and data access patterns. Both *particlefilter* and *bfs* have one array that Porple puts into the constant memory, whereas the rule-based method puts it into global memory. Program *spmv* has one array *vec* that is used many times. By analyzing the data access patterns, Porple discovers that there will be many cache hits if that array is put into the texture memory. For its array *cols,* Porple also puts it into the texture memory, because its path-based performance model recognizes that even though the array could have similar access latency on global memory, putting it into texture memory works better, because some arrays are already put into the shared memory and thus the path to global memory is busy. The placements lead to 69 percent performance improvement. The rule-based approach places all arrays on texture memory, causing large contention. The situation for *md* is similar. For *cfd,* the rule-based approach places all read-only arrays onto the texture memory, improving performance by 64 percent. Porple tries to balance the bandwidth used by global memory and texture memory. It puts three arrays onto the texture memory, getting 70 percent performance improvement.

The path-based performance model outperforms a latency-based model that we used before.[4] Thanks to its advantage in balancing bandwidth usage among the paths, the new model helps improve the speedups from 2.43 to 2.64 times on *trans,* 1.62 to 1.69 times on *spmv,* and 1.62 to 1.7 times on *cfd.* Table 3 compares the bandwidth utilization measured through hardware performance counters. For *trans,* because there is no data reuse in the kernel, a higher bandwidth utilization of L2 cache and device memory results in a better performance of the program. For *spmv* and *cfd,* there are many data reuses during the kernel execution. The higher bandwidth utilization of the L1 cache and texture cache suggests better cache performance.

Figures 4b and 4c show the speedups on M2075 and C1060. Porple again significantly outperforms the rule-based method. Table 4 offers the details on the Porple-selected data placements and the rule-based method on the three generations of GPU. The rule-based approach uses the same data placement decision for a program on all platforms. In contrast, Porple can tailor the data placements to each platform, producing much better performance.

Regular benchmarks, in general, have less potential for enhancement due to their simplicity. The rule-based approach provides an average 5 percent performance improvement, whereas Porple provides an average 28 percent improvement, which is 1 percent away from the upper bound. For the memory-intensive benchmark *trans,* Porple achieves 2.64 times speedup by placing array *odata* in surface memory and array *idata* in texture memory.

Runtime overhead mainly comprises the time to collect memory access patterns, search for the best placement, and select the version. Similar to the original Porple,[4] the overall

Table 3. Bandwidth usage for *trans*, *spmv*, and *cfd*. Results are given in Gbytes per second.

Benchmarks	*trans*				*spmv*				*cfd*			
Version	L1/ shared	L2	Texture	Device	L1/ shared	L2	Texture	Device	L1/ shared	L2	Texture	Device
Original	114.174	114.188	0	90.245	284.737	263.16	0	91.954	153.634	217.518	9.557	30.59
Rule based	114.174	114.188	0	90.245	34.447	324.096	308.734	143.707	4.948	348.236	152.383	58.694
Porple-latency	58.823	176.47	0	146.64	165.542	558.821	105.806	146.341	4.809	393.905	148.107	58.155
Porple	0	199.267	132.844	150.353	105.943	578.487	221.979	154.85	101.764	420.548	73.207	57.068

*L1/shared: L1 cache and shared memory; L2: L2 cache; Texture: texture cache; Device: device memory.

Table 4. Placement decisions made by Porple and the rule-based approach.

	spmv					*particlefilter*					
Decision maker	A0	A1	A2	A3	A4	B0	B1	B2	B3	B4	B5
Rule-based approach	T	T	T	T	G	G	S and G	G	G	G	G
Porple-M2075	C	G	T	G	G	C	S and G	T	T	G	G
Porple-K20c	C	T	T	G	G	C	S and T	T	T	G	G

*T: texture memory, C: constant memory, G: global memory, S: shared memory. In *spmv*, A0: rowDelimiters, A1: *cols*, A2: *vec*, A3: *val*, A4: *out*. In *particlefilter*, B0: CDF, B1: u, B2: arrayX, B3: arrayY, B4: xj, and B5: yj.

overhead never exceeds 5 percent of the total kernel execution time. It is included in the reported speedup results. The experiments are all on CUDA benchmarks on Nvidia devices because the current Porple implementation is CUDA-based. However, the method can apply to OpenCL and other GPUs. For instance, after manually applying the transformation to the *spmv* program in the OpenCL software development kit, we saw 1.76 times speedups on K20c. Fully porting Porple to OpenCL is left to future work.

Good system software support is essential for translating carefully designed memory systems into actual program performance. The Porple system in this article demonstrates a promising solution. Its use of MSL helps separate hardware complexities from other concerns, enabling cross-device portability of the optimization. Its carefully designed compiler-and-runtime synergy helps address input sensitivity of data placements. Future memory systems are expected to turn even more sophisticated and complex, evidenced by the various emerging memory technologies (such as phase change and stacked memory). The Porple approach, for its distinctive advantages on portability, could open up many other opportunities for tapping into the full power of future memory systems. MICRO

References

1. B. Jang et al., "Exploiting Memory Access Patterns to Improve Memory Performance in Data-Parallel Architectures," *IEEE Trans. Parallel and Distributed Systems*, vol. 22, no. 1, 2011, pp. 105–118.
2. Y. Zhang and F. Mueller, "Auto-generation and Auto-tuning of 3D Stencil Codes on GPU Clusters," *Proc. 10th Int'l Symp. Code Generation and Optimization*, 2012, pp. 155–164.

3. S. Hong and H. Kim, "An Analytical Model for a GPU Architecture with Memory-Level and Thread-Level Parallelism Awareness," *Proc. 36th Ann. Int'l Symp. Computer Architecture*, 2009, pp. 152–163.
4. G. Chen et al., "Porple: An Extensible Optimizer for Portable Data Placement on GPU," *Proc. 47th Ann. IEEE/ACM Int'l Symp. Microarchitecture*, 2014, pp. 88–100.
5. A. Danalis et al., "The Scalable Heterogeneous Computing (SHOC) Benchmark Suite," *Proc. 3rd Workshop General-Purpose Computation on Graphics Processing Units*, 2010, pp. 63–74.
6. S. Che et al., "Rodinia: A Benchmark Suite for Heterogeneous Computing," *Proc. IEEE Int'l Symp. Workload Characterization*, 2009, pp. 44–54.

Guoyang Chen is a PhD student in the Department of Computer Science at North Carolina State University. His research interests include heterogeneous computing, compilers, and parallel computing. Chen has a BS in computer science from the University of Science and Technology of China. Contact him at gchen11@ncsu.edu.

Bo Wu is an assistant professor in the Department of Electrical Engineering and Computer Science at the Colorado School of Mines. His research interests include high-performance computing, heterogeneous computing, program analysis and optimizations, and compilers. Wu has a PhD in computer science from the College of William and Mary. Contact him at bwu@mines.edu.

Dong Li is an assistant professor in the Department of Electrical Engineering and Computer Science at the University of California, Merced. His research interests include high-performance computing, performance modeling, programming models, architecture, and runtime. Li has a PhD in computer science from Virginia Tech. Contact him at dli35@ucmerced.edu.

Xipeng Shen is an associate professor in the Department of Computer Science at North Carolina State University. He is also an adjunct associate professor at the College of William and Mary and an IBM Canada CAS Research Faculty Fellow. His research interests include compilers, program analysis and optimizations, heterogeneous computing, high-performance computing, and machine learning. Shen has a PhD in computer science from the University of Rochester. Contact him at xshen5@ncsu.edu.