Enabling Multiple Accelerator Acceleration for Java/OpenMP

Ronald Veldema, Thorsten Blass, Michael Philippsen
University of Erlangen-Nuremberg, Computer Science Department
Programming Systems Group, Erlangen, Germany
veldema@cs.fau.de, thorsten.blass@cs.fau.de, philippsen@cs.fau.de

***Abstract*—While using a single GPU is fairly easy, using multiple CPUs and GPUs potentially distributed over multiple machines is hard because data needs to be kept consistent using message exchange and the load needs to be balanced. We propose (1) an array package that provides partitioned and replicated arrays and (2) a compute-device library to abstract from GPUs and CPUs and their location. Our system automatically distributes a parallel-for loop in data-parallel fashion over all the devices.**

There are three contributions in this paper. First, we provide transparent use of multiple distributed GPUs and CPUs from within Java/OpenMP. Second, we partition arrays according to the compute-devices' relative performance that is computed from the execution time of a small micro benchmark and a series of small bandwidth tests run at program start. Third, we repartition the arrays dynamically at run-time by increasing or decreasing the number of machines used and by switching from CPUs-only to GPUs-only, to combinations of CPUs and GPUs, and back. With our dynamic device switching we minimize communication while maximizing device use.

Our system automatically finds the optimal device sets and achieves a speedup of 5 – 200 on a cluster of 8 machines with 2 GPUs each.

***Keywords*-Parallel computing, Java, GpGPU, OpenMP**

I. Introduction

OpenMP [1] is well-suited for shared-memory parallel processing. There are of course implementations on top of SMPs, NUMAs and Distributed Shared Memory (DSM) systems, but not much work has been done on targeting OpenMP for heterogeneous distributed architectures. We propose to use OpenMP even for distributed architectures because shared memory programming is still far easier than distributed programming. This is even more the case when not all processors are the same.

ClusterJaMP is a version of OpenMP for Java [2], [3]. It provides the common OpenMP annotations for translation to multi-core. jCudaMP [2] already translated parallel-for statements into Cuda code, however, it (like its ilk) can only use a single GPU comfortably, because both OpenMP and Java assume a single global address space which is no longer present in a multi-machine and/or multi-GPU environment (each GPU also has its own memory and address space).

When only a single GPU is used, the shared memory illusion can be implemented by copying all required data to the GPU, waiting for it to finish, and copying the data back. This provides an illusion of a single path of control and a single address space. But this simple model does no longer hold when we try to use multiple GPUs (what to copy where?) and/or multiple machines. In general, when using multiple compute devices (including the CPUs), there are a few problems that need to be solved:

1 What compute devices are available and how fast are they? How fast can data be transported to them?
2 How to manage data consistency across multiple address-spaces?
3 How to spread the computational load in a fair way across compute devices of different speeds and communication latencies?
4 How many compute devices and/or machines to use for optimal performance?

In this paper we address (1) by providing an abstraction layer that abstracts from both the network and the concrete device implementation (Cuda, OpenCL, etc.). We solve (2) by providing distributed array classes. An array class is then responsible for maintaining its consistency. We solve (3) and (4) by running an initial separate benchmark to probe each discovered device's performance parameters as well as the network latency and bandwidth to that device's memory. The benchmark results are used to balance the load equitably across the devices.

Additionally, we can redistribute the work at run-time by increasing (or decreasing) the number of devices and even switch between GPU, CPU, and CPU+GPU device types to locate the sweet spot of the application performance. Decreasing the number of machines or devices used can improve performance by decreasing the amount of communication. However, if the network is not yet saturated and the devices are fully loaded, using more compute devices can also increase performance.

Note that to take best advantage of dynamically changing the number of used processors it is necessary to change the cluster's management software. Ideally, the cluster's software should dynamically change process groups to maximize cluster utilization.

Our contributions are:

1 transparent use of *multiple* distributed CPUs and GPUs from Java and OpenMP;
2 an array partitioning that takes individual accelerator speeds into account;

```
void foo(int N) {
  //#omp managed(a)
  int[]a = new int[N];

  //#omp parallel for
  for (int i=0;i<N;i++) {
    a[i]++;
  }
}
```

Figure 1: Example ClusterJaMP program.

3 an adaptive tuning of the set of compute devices (by increasing or decreasing the number and selecting the type of accelerators) that finds an optimal communication/computation ratio for best performance.

II. ClusterJaMP

We study the problem of multi-GPU and multi-machine use in the context of an OpenMP adapted to Java, called ClusterJaMP. Without a loss of generality, all our results should be transferable to OpenMP for C++/Fortran.

ClusterJaMP's compilation pipeline is as follows. First, the programmer annotates the source code with OpenMP annotations. The annotated code is fed to ClusterJaMP's compiler which is implemented as an extension to the Java compiler of Eclipse. The resulting Java bytecode files are annotated with the locations of each parallel section and their OpenMP attributes. At run-time, the standard Java class loader is replaced by the ClusterJaMP class loader. For each loaded class file it checks if it holds the OpenMP annotations and if so, it passes control to the Compute Device Manager (CDM) to JIT-compile each parallel-for in a device specific way (CDM is described in Sec. III). This compilation process creates 'kernels', functions that contain the parallel loop's body and are parametrized by all live variables. Any regular code without OpenMP annotations is passed to the standard Java VM's JIT compiler.

To enable the use of multiple accelerators at the same time and to exploit their computation power efficiently, we slightly extend the current set of OpenMP directives. The new directives help the compiler (1) to optimizing the partitioning process, (2) to replace Java arrays with array packages so that arrays can be managed by ClusterJaMP and, (3) to enable support for parallelization of perfectly nested loops to expose more parallelism. We cannot show (3) in this paper due to space restrictions.

In Fig. 1 we show a simple ClusterJaMP program. Because the array `a` is used in a parallel context, the compiler requires that `a` is marked with `managed`. This causes the array to be allocated on the accelerators. For a managed array the ClusterJaMP compiler rewrites the code and replaces the Java array with a proxy object to the native C++ object that performs the actual array management. This involves replacing any managed array with its proxy type. To

```
// Java:
void foo(int N) {
  IntArray1D a = new IntArray1D(N);
  JNI_call(do_parallel_for, a.get_C_ptr(), N);
}
// C++:
void do_parallel_for(IntArray1D *a, int N) {
  argument_list args = allocate_argument_list();
  args.add(a);
  args.add(N);
  cdm_parallel_for(0, N, args, kernel_of_foo);
}
// OpenCL/CDM-C kernel:
void kernel_of_foo(int *a) {
  int i = get_loop_index_1d();
  a[i]++;
}
```

Figure 2: Pseudocode of generated ClusterJaMP program.

```
//#omp alloc_start
   //#omp managed(a)
   int []a = new int[N];
   //#omp managed(b)
   int []b = new int[M];
//#omp alloc_end
```

Figure 3: Example data allocation block.

simplify compilation, it is not allowed for a managed array to have pointers to non-managed arrays or objects. This ensures that an array will never have pointers to machine-local data.

The compiler transforms the example code from Fig. 1 to that of Fig. 2. It replaces the `int[]` type with `IntArray1D` in `foo` and places the loop's body into a new OpenCL/CDM-C function `kernel_of_foo()`.

One problem when using multiple compute devices is how to reasonably partition the data of multiple individual arrays. If this issue was ignored, one big array could completely fill a device's memory leaving no room for the partitions/replicas of arrays instantiated later. Our solution is to allow a programmer to group his array allocations in a block. This is shown in Fig. 3. The ClusterJaMP compiler then allocates both arrays `a` and `b` in one step and in cooperation with each other.

III. Compute Device Manager, CDM

A cluster may hold different compute devices. Instead of targeting each accelerator type (CPU, GPU, Cell, etc.) and API (Cuda, OpenCL, etc.) separately, we create a new abstraction layer "Compute Device Manager" (CDM). It provides an abstract `ComputeDevice` class that encapsulates in its current version the Cuda, OpenCL, and the main (multi-core) CPU devices. Cuda is a C dialect that adds a parallel-call statement to invoke a kernel multiple times in parallel on a GPU. OpenCL is Cuda-like, but standardized. For the main multi-core CPU we create a ComputeDevice

that starts one thread per core. Device specific optimizations are left to the devices' JIT or normal compilers.

In short, CDM's abstract `ComputeDevice` class provides methods that

- allocate and free memory on a specific device,
- invoke a method on a specific device, and that
- provide relative speed and memory size information.

If an abstract device proxies some remote hardware, MPI [4] is used for message exchange.

After initialization, all devices on all machines are quickly benchmarked, all with the same kernel, so that CDM can compute each device's relative speedup compared to the slowest device.

IV. Cluster Array Package

The (Java-OpenMP) compiler rewrites arrays used inside of parallel regions to use one of the classes from our Cluster Array Package (CAP) instead. The compiler has the choice between a partitioned array and a replicated array which can then be further specialized for their element types (int, float, double, etc.). The partitioned array class distributes array elements over a given set of devices. The replicated array class replicates its data over all the devices in a device set. Both classes use CDM's abstract `ComputeDevice` classes to access the real hardware.

A. Partitioned Array Class

Partitioned arrays distribute subsets of array indices over a given set of devices. The higher the computing power of a device is, the larger is its chunk and hence its parallel workload. We allow a Java array to be implemented by means of a partitioned array if:

(1) there are no data dependencies between different loop iterations on read/write accesses on this array *and*
(2) the indexing functions only add or subtract a small constant from the loop variable.

If either of these constraints is violated, we must resort to using a replicated array for that Java array. Of course, a replicated array has the disadvantage that it occupies far more memory on the cluster than a partitioned array does.

To simplify matters we only partition the single highest array dimension instead of creating tiles or blocks. This reduces algorithm complexity, allows for easier creation of bulk transfers, and avoids fragmentation of the index set. If a kernel contains both an access to `a[i]` and `b[i]`, where both arrays can be partitioned, both arrays are partitioned in the same way. This way no kernel will ever have to access remote partitions. In particular, partitioned arrays can be used with "stencils" whose ghost cells will automatically be created and updated.

B. Replicated Array Class

If a Java array cannot be implemented as a partitioned array, it must be implemented by means of a replicated array. Unfortunately, random writes to a replica trigger several steps to restore data consistency among all replicas. After all kernels on all devices have finished, our replicated array package merges all the changes that have been made to the replicas and propagates them to all other replicas of the array. This is done using the following steps.

1) Each machine stores a single additional local copy of the array (called a twin) in its main memory. This copy is allocated when the replicated array is instantiated.
2) After a kernel finishes, for all devices in a machine (including the CPUs), we create (in parallel) a diff-vector using the twin from (1). The diff-vectors are exchanged between all machines.
3) Each machine updates the replicas on all its devices by applying all incoming diffs, including the diff created locally if a machine holds multiple compute devices. It is considered a program error if more than one processor updates a variable at a time.

V. Adaptively changing the number and types of devices used

A central issue in cluster computing is to select the optimal number of cluster nodes to use for an application to get a good computation/communication ratio. On heterogeneous cluster nodes, the device types also need to be selected. For example, a data-flow dominated (streaming) kernel will perform better on a GPU, while a kernel that is dominated by control-flow or random memory access benefits from the branch prediction or caching of the CPU. Autotuning relieves the programmer of having to manually select the set of devices to use. Note that growing/shrinking assumes that the work load is proportional to the data size. It is open research to find a load distribution for other types of problems.

We test two autotuning heuristics in this paper: "grow" and "shrink". Reconfiguration of the cluster is triggered on every n^{th} parallel invocation for the following parallel invocations. Our autotuner does not get stuck in local minima by detecting that a change (more or fewer nodes or a different device type) only causes a slight improvement or even a loss of performance. In this case another device type (or combination thereof) is tried.

Grow-heuristics. We start with one machine and a device of some type. Then a reconfiguration doubles the number of machines used for a parallel-for. Whenever we double the number of machines, we add the newly available devices of the same device type to the set that executes the next parallel-for. Once all machines are used, we switch to using another device type and start from scratch using only one machine and one device of that new type.

Table I: Application performance (in seconds) using a fixed configuration (bold numbers highlight best performance).

	1 CPU	2 CPUs	4 CPUs	8 CPUs
Edge Detection	6.6	3.7	1.3	0.7
Matrix Multiply	178.5	97.8	50.6	26.5
M.M. + Nest	175.2	95.1	49.3	25.8
Black Scholes	142.3	113.7	61.7	37.4
SFE (Constructor)	254.8	135.6	65.7	35.8
	2 GPUs	4 GPUs	8 GPUs	16 GPUs
Edge Detection	2.90	0.90	0.04	**0.02**
Matrix Multiply	121.30	64.40	33.00	20.30
M.M. + Nest	5.80	3.10	1.60	**0.80**
Black Scholes	0.86	0.67	0.52	**0.45**
SFE (Constructor)	17.30	9.30	4.70	**1.60**
	1 CPU + 2 GPUs	2 CPUs + 4 GPUs	4 CPUs + 8 GPUs	8 CPUs + 16 GPUs
Edge Detection	2.9	0.9	0.2	0.09
Matrix Multiply	104.3	53.8	29.0	**18.80**
M.M. + Nest	22.7	13.2	8.0	4.80
Black Scholes	14.2	11.7	9.6	5.40
SFE (Constructor)	33.3	17.9	8.2	3.20

Once all device types have been tried, we start trying the combination of two device types (CPU and GPU, for example).

Shrink-heuristics. We start with a device set containing all available devices of all machines. On each reconfiguration we shrink by halving the number of machines and by then ignoring the devices from the removed machines. Once only a single machine is left, we restart with a full set and remove a different device type.

Reconfiguration is not cheap as array chunks have to be copied between devices to fit both the new configuration and the corresponding work distribution. With both heuristics, if a machine is added or removed all partitioned arrays must be repartitioned. For replicated arrays, a replica is allocated for each new device that is newly included into the configuration.

VI. Performance

All measurements were performed on a cluster consisting of 8 machines, each with two Xeon 5550 "Nehalem" chips (8 cores + hyperthreading) running at 2.66 GHz with 8 MB shared cache per chip, 24 GB of RAM (DDR3-1333). Each machine holds two nVidia Tesla M1060 GPUs (4 GByte of memory each) and an Infiniband interconnect with 20 GBit/s bandwidth per link and direction. Each node runs Linux kernel 2.6.18 and all benchmarks were compiled with a combination of Cuda 3.1 and GNU G++ 4.3.3. For inter-node communication, we rely on OpenMPI [4] version 1.4.2. To measure the shrink/grow versions, we submit a fixed size reservation to Torque/Maui [5] and then sub-allocate processors within the reservation. Since the data used in the benchmarks is mostly too large for a single GPU's memory, there are no single-GPU times.

To stress-test the cluster's network performance, we examined the achievable inter-device bandwidths by copying chunks of main-memory from the primary machine to another device's memory, potentially on another machine. We found that for copying inside a machine, bandwidth from one device to another is around 4 GByte/sec. When copying to another machine's device, the copying bandwidth drops to around 590 MByte/sec, regardless of CPU → CPU or CPU → GPU.

The application measurements are shown in Table I. The applications' performance with the adaptive reconfiguration autotuner switched on is shown in Table II. All measurements are performed five times and averaged. Slight jitter occurs due to asynchronous message processing (a thread is awoken from a pool to handle an incoming message). Overall, when adaptive configuration was enabled, the system always automatically selected the GPUs only. The only exception was the naive matrix multiply where all available CPUs and all their GPUs were finally selected.

Without looking closer at the benchmarks themselves, we can make some global conclusions. First, each line of Table I shows that speedup is good regardless of the device type combination. However, using a combination of CPU and GPU does not achieve extra performance as the amount of data handed to the CPU is vanishingly small. Second, dynamic reconfiguration clearly works and is surprisingly cheap: in 7 to 11 reconfigurations an optimum is found as Table II shows. Although a single benchmark that cannot be representative for all applications is used to guide initial data placement, the heuristics always find the configurations that yield the best performance (most often a GPU-only configuration). This is due to the GPUs being far faster than the CPUs so that the CPUs have proportionally little to nothing to do under our naive compilation model. In all cases, the "grow" heuristics had to try more device sets (8 to 11 vs. 7 to 8) to find the optimal configuration (see Table II). The reason is that "grow" has fewer opportunities to quickly abort and backtrack as it slowly climbs to better performance. The "shink" heuristics, in contrast, starts with good performance and quickly backtracks when seeing worse performance compared to the initial GPU+CPU combination. The small differences in final run-time performance between the times in Table I and Table II are due to the message passing library and threading system jitter. A single tuning round takes in the order of 0.1 to 0.5 milliseconds and does not seriously affect the total run time.

Finally, all applications needed the allocation block extension (Sec. II) to cooperatively allocate individual arrays. Without the annotation, the applications would not work.

A. Edge detection

Edge detection is a fundamental image processing problem. Edges in images are characterized by an intensity change from one pixel of an image to the adjacent pixel.

Table II: Application performance (in seconds) using adaptive reconfiguration.

GROW	best time grow (sec)	# tried grow sets
Edge Detection	0.02	9
Matrix Multiply	19.20	11
M.M. + Nest	0.78	10
Black Scholes	0.43	10
SFE (Constructor)	1.90	8
SHRINK	best time shrink (sec)	# tried shrink sets
Edge Detection	0.02	8
Matrix Multiply	19.40	7
M.M. + Nest	0.74	7
Black Scholes	0.46	7
SFE (Constructor)	1.70	7

The Sobel edge detector [6] uses a gradient approximation. The superlinear speedup we see in this benchmark is because of the memory bandwidth bottleneck that is overcome when adding more GPUs.

The compiler detects that the image can be implemented and spread over the devices by using a partitioned array class. Both "grow" and "shrink" heuristics (Table II) find the same optimal device set (which they do for all applications tested modulo a little application performance jitter).

B. Matrix Multiply

This benchmark implements a naive (non-blocking) matrix multiplication. We use 5000×5000 matrices of integer elements and perform the matrix multiplication in a loop 100 times. We also measure performance with OpenMP's collapse clause applied that collapses the two nested loops into a single larger loop. See the lines in the tables marked with 'M.M. + Nest'.

All matrices are stored as replicated arrays as (from the point of view of the compiler) random access occurs to each. After each iteration we therefore need to unify the matrix replicas (Sec. IV-B) by broadcasting diffs both within and between machines.

Speedup is good in all cases as the network bandwidth is not exhausted. Without nested loops, the system shows a slight advantage to using both CPUs and GPUs simultaneously (as the GPUs are not fully loaded).

Adaptive reconfiguration of the non-nested version finds the optimal device set at 8 CPUs and 16 GPUs after only 7 shrink steps.

C. Black Scholes

The Black Scholes model [7] uses a partial differential equation to predict the prices of European options. This benchmark calculates the prices for an option-call and option-put in parallel. All data is stored in one dimensional, partitioned float arrays of the same size. The prices are calculated in a single parallel-for loop.

The application shows a reasonable speedup on the CPUs (3.8 on 8 CPUs). Going from 2 GPUs to 8 GPUs shows less speedup (1.8) as the network latency starts to hinder performance over the short run-times. Again adaptive reconfiguration finds the optimal device set.

D. Secure Function Evaluation (SFE)

SFE [8] is a cryptographic protocol that enables the secure evaluation of an arbitrary boolean circuit between several parties on private inputs. SFE has high memory and compute requirements for creating and evaluating of a circuit.

SFE shows good speedup when using CPUs (7.1 using 8 machines) and when going from 2 GPUs to 8 GPUs (a speedup of 10.1). The combination of CPUs and GPUs is slower than only using GPUs because the CPU receives chunks that are too big compared to their SFE performance. This could potentially be avoided by running multiple benchmarks at program start-up to measure relative device performances. The benchmark that is most similar to the program's kernels should then dictate array partitioning.

VII. Related Work

A number of new languages have recently been proposed for high-performance, high-productivity computing, for example X10 [9] and Chapel [10]. To increase performance, these languages have built-in support for locality but are thus far unable to run in environments with several different compute devices. In this paper we propose minimal language extensions (OpenMP) and a combination of array classes and compiler analysis to allow efficient, locality aware computing. The ideas in this paper can of course also be applied to implementations of these new languages.

A number of OpenMP implementations allow transformation to Cuda code, e.g. [11], [12]. Neither of these, however, supports multiple GPUs or multiple machines i.e. clusters. Their focus is instead either on supporting the full OpenMP standard or on generating optimized Cuda code. Our system instead focuses on multi-GPU and multi-machine use combined with finding optimal sets of devices to use.

Zippy [13] ports the Global Arrays programming model to allow GPU use. A global array is a distributed array that partitions its elements over multiple GPUs. The array object then has a fixed set of methods to operate on the arrays. Our system abstracts from the array objects and allows the programmer to write normal Java code. With our adaptive optimization the system dynamically finds the near optimal set of devices to use and also partitions arrays automatically.

Instead of array packages that span all compute devices, Cudasa [14] provides a DSM where the GPUs are explicitly programmed. Here the Cuda programming model is extended with task annotations to allow cluster computing. GPUs and tasks have to be explicitly programmed in Cudasa. Our system abstracts from tasks and GPUs completely and creates a full DSM.

In [15] an asymmetric DSM system is presented where the host CPU can page-in GPU memory on demand (a host's page fault translates to a page allocation and a GPU-to-host memory copy). The asymmetry is that the GPUs cannot page-in host memory to their memory. Our system does not require such explicit GPU $\leftrightarrow$ CPU copies.

Conclusion

We have described a system that (1) allows transparent use of multiple GPUs and machines from OpenMP, (2) partitions arrays based on accelerator speed, and (3) dynamically adapts the set of compute devices to reach an optimal configuration of CPUs and GPUs. On a cluster with 8 CPUs and 16 GPUs we archive good speedups (up to 200x compared to the performance of a single CPU). Reconfiguration requires only a few (7–8) autotuning iterations to find the optimal set using a "shrink" heuristics which is better than "grow". Some small benchmarks run at program start-up provide a good basis for device specific data partitioning.

References

[1] OpenMP Architecture Review Board, *OpenMP C and C++ Application Program Interface, Version 2.0*, March 2002. [Online]. Available: http://www.openmp.org/

[2] G. Dotzler, R. Veldema, M. Klemm, and M. Philippsen, "jCudaMP: OpenMP/Java on CUDA," in *Third Intl. Workshop on Multicore Software Engineering (IWMSE10)*, Cape Town, South Africa, May 2010, pp. 10–17.

[3] Michael Klemm, Matthias Bezold, Ronald Veldema, and Michael Philippsen, "JaMP: An Implementation of OpenMP for a Java DSM," in *Proc. 12th Workshop on Compilers for Parallel Computers (CPC)*, A Coruna, Spain, Jan. 2006, pp. 242–255.

[4] E. Gabriel, G. E. Fagg, G. Bosilca, T. Angskun, J. J. Dongarra, J. M. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine, R. H. Castain, D. J. Daniel, R. L. Graham, and T. S. Woodall, "Open MPI: Goals, concept, and design of a next generation MPI implementation," in *Proc. 11th European PVM/MPI Users' Group Meeting*, Budapest, Hungary, Sept. 2004, pp. 97–104.

[5] D. B. Jackson, Q. Snell, and M. J. Clement, "Core Algorithms of the Maui Scheduler," in *Revised Papers from the 7th Intl. Workshop on Job Scheduling Strategies for Parallel Processing*, ser. JSSPP '01, Cambridge, MA, June 2001, pp. 87–102.

[6] R. C. Gonzalez and R. E. Woods, *Digital Image Processing*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1992.

[7] F. Black and M. Scholes, "The pricing of options and corporate liabilities," *Political Economy*, vol. 81, no. 3, pp. 637–54, 1973. [Online]. Available: http://ideas.repec.org/a/ucp/jpolec/v81y1973i3p637-54.html

[8] A. C. Yao, "How to generate and exchange secrets," in *IEEE Symp. on Foundations of Computer Science (FOCS'86)*, Toronto, ON, Canada, Oct. 1986, pp. 162–167.

[9] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar, "X10: an object-oriented approach to non-uniform cluster computing," in *Proc. Conf. on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'05)*, San Diego, CA, Oct. 2005, pp. 519–538.

[10] B. L. Chamberlain, D. Callahan, and H. P. Zima, "Parallel Programmability and the Chapel Language," *Intl. J. High Perform. Comput. Appl.*, vol. 21, no. 3, pp. 291–312, 2007.

[11] S. Ohshima, S. Hirasawa, and H. Honda, "OMPCUDA: OpenMP Execution Framework for CUDA Based on Omni OpenMP Compiler," *Lecture Notes in Computer Science*, vol. 6132, pp. 161–173, 2010.

[12] S. Lee, S.-J. Min, and R. Eigenmann, "OpenMP to GPGPU: a compiler framework for automatic translation and optimization," in *Proc. 14th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming (PPoPP '09)*, Raleigh, NC, Feb. 2009, pp. 101–110.

[13] Z. Fan, F. Qiu, and A. E. Kaufman, "Zippy: A Framework for Computation and Visualization on a GPU Cluster," *Comput. Graph. Forum*, vol. 27, no. 2, pp. 341–350, 2008.

[14] M. Strengert, C. Müller, C. Dachsbacher, and T. Ertl, "CUDASA: Compute Unified Device and Systems Architecture," in *Eurographics Symp. on Parallel Graphics and Visualization (EGPGV08)*, Crete, Greece, April 2008, pp. 49–56.

[15] I. Gelado, J. E. Stone, J. Cabezas, S. Patel, N. Navarro, and W.-m. W. Hwu, "An Asymmetric Distributed Shared Memory Model for Heterogeneous Parallel Systems," *SIGARCH Comput. Archit. News*, vol. 38, no. 1, pp. 347–358, 2010.