Object support for OpenMP-style programming of GPU clusters in Java

Carolin Wolf, Georg Dotzler, Ronald Veldema, Michael Philippsen
University of Erlangen-Nuremberg, Computer Science Department
Programming Systems Group, Erlangen, Germany
<carolin.h.wolf|georg.dotzler|ronald.veldema|michael.philippsen>@fau.de

***Abstract*—For scientists, it is advantageous to use a high level of abstraction for programming their simulations, so that they can focus on the problem at hand instead of struggling with low-level details. However, current HPC clusters with multiple GPUs per node only offer explicit communication to and from the GPUs, require manual work to keep the data consistent, and often need explicit kernel programming. Moreover, known GPU programming frameworks are limited to a single GPU or a single machine and also rarely support objects.**

Our system removes the above restrictions. With a slight but necessary change in Java's semantics, we achieve automatic distribution and efficient use of objects and arrays of objects on multiple GPUs in a cluster.

On benchmarks that distribute arrays of objects over five machines with 10 GPUs, we achieve speedups of up to 4.9 compared to one node.

***Keywords*-Java; Parallelism; GPU; Cluster computing; OpenMP**

I. Introduction

Today's clusters often have a combination of multi-core CPUs and 1-4 GPUs per node. Whereas CPUs only have a few cores, are optimized for general purpose programming and are usually programmed with a combination of C/C++/Fortran and MPI (Message Passing Interface), GPUs have many cores (1000+) and are programmed in dialects of C/C++ with parallel extensions. For best performance in GPU applications, data should be bulk transferred to the individual GPUs, which further adds to the complexity of the necessary code. This heterogeneous and low-level programming model limits productivity. Hochstein et al. show in their case study [1] that around 20% of the development effort of a HPC program is spent on MPI-related issues alone. Christadler et al. [2] compare parallel programming languages and also find that the use of MPI in combination with CUDA results in considerably more lines of code and reduced programmer productivity. The study also compares X10 [3] and Chapel [4] that both represent the address space of each compute device in a first-class data structure ('place' or 'locale'). Although both languages lead to a shorter development time, their performance falls behind the other programming languages. Other studies, like the one from Wienke et al. [5], also show that programs written in CUDA lead to higher speedup but increase the effort of the developers.

Our solution is to extend the shared memory programming model of OpenMP [6] to a distributed memory model that not only spreads the iteration spaces of OpenMP parallel-for loops across *all* available GPUs and CPUs, but also efficiently supports not just arrays of primitive data types, but also arrays of objects in parallel regions.

Object-orientation lets a programmer write code in a far more intuitive way, without the need to focus on how the computation is distributed across the system, and where to place the needed data. However, objects cause new problems:

- Methods have to be compiled for both the CPUs and the GPUs.
- Kernels have to be executed on exactly the devices that store the array chunks needed for their computation.
- Data transfers must deep-copy objects (objects can contain references to other objects).

We use the Java programming language, because many features from Java, for example its safety features and the garbage collector, reduce the programmers' effort to develop the sequential code. The optimization of the generated CUDA code is also easier, as there is no pointer arithmetic that requires an additional analysis, like e.g. in C++. Finally, many prospective HPC programmers are already experienced in Java, but have no thorough knowledge of Fortran.

Our main contribution is a programming model that has to diverge slightly from Java's semantics in parallel-for loops, but that allows efficient object-oriented data-parallel programming of clusters with many GPUs.

Below, we first explain our software stack. Section III describes how we handle objects in a parallel-for loop. Section IV presents the code generation, data management and restrictions that have to be made on the use of objects. Section V covers the related work, and Section VI discusses the performance of our system for a number of benchmarks before we conclude.

II. Architecture

In our Java/OpenMP extension, called JaMP, an OpenMP-style parallel-for directive [6] indicates that a for-loop can be executed not only on the CPU in parallel, but also on clusters with GPUs. The implementation of the JaMP compiler is built on the Eclipse Java compiler, see Fig. 1.

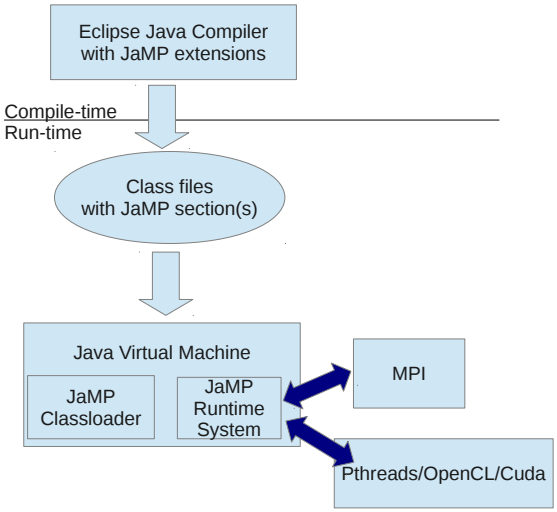

Figure 1. Architecture of the JaMP system.

It inspects parallel-for loops and extracts information about the occurring methods and data types. In order to represent the classes for the kernels, the compiler generates C-style struct declarations and functions to copy objects between the JVM (Java Virtual Machine) and these structs. The structs and functions are stored in special attributes of the resulting Java bytecode files. Arrays (called managed arrays below) are distributed over multiple nodes if necessary.

In addition, the JaMP compiler creates a native library for every data type that appears as managed object array, see Sec. III. This native library is responsible for accessing the distributed arrays' data through Java's JNI (Java Native Interface) at run-time. The compiler transforms every access to a managed array element into a get/set method call. The library knows and hides the location of the array elements on the CPUs and GPUs.

The middleware ('JaMP Runtime System') generates and invokes the compute kernels in parallel on all devices. It abstracts from the types of available compute devices and their locations in a cluster. It also provides transparent remote references to both memory and compute kernels, and consists of three layers (all written in C++):

1) The **StreamRPC** (SRPC) layer offers a low-level API for remote copy requests and procedure calls. This layer uses MPI.
2) The **ComputeKernelManager** layer (CKM) uses SRPC to manage kernels. It is responsible for creating and compiling the kernels in C++, OpenCL and CUDA, and also for starting them, regardless of the locations of the accelerators.

```
1 class FloatVector3D {
2   float x, y, z;
3   FloatVector3D(float xval, float yval,
4                 float zval) {
5     x = xval; y = yval; z = zval;
6   }
7   void add(FloatVector3D other) {
8     x += other.x;
9     y += other.y;
10    z += other.z;
11  }
12 }
13
14 public static void main(String[] args) {
15   FloatVector3D delta
16     = new FloatVector3D(1.0f, 2.0f, 3.0f);
17   //#omp managed(vectors)
18   FloatVector3D[] vectors
19     = new FloatVector3D[1024];
20
21   //#omp parallel for
22   //&omp     shared(vectors, delta)
23   for (int i = 0; i < 1024; i++) {
24     vectors[i].add(delta);
25   }
26 }
```

Figure 2. Example of a vector addition program in JaMP.

3) The **ClusterArrayPackage** layer (CAP) organizes CKM jobs for managed arrays. For any computation on a managed array, a kernel is started on each of the devices that store a portion of the array data. In addition, CAP uses CKM's copying functionality to send objects to these devices if needed. CAP furthermore abstracts from the CKM compute devices to present a uniform data-parallel access layer.

Altogether, this middleware hides the details of a heterogeneous multi-GPU cluster, and turns it into a set of abstract compute devices with a data-parallel way to access arrays.

III. Programming Model

In our framework, non-array Java objects behave like standard OpenMP *shared objects*, whereas *managed arrays* are treated differently. Fig. 2 shows an example program with both types. The variable `delta` (line 15) is a simple shared object and `vectors` (line 18) is a managed array of `FloatVector3D` elements.

JaMP *shared objects* can be read and written by each GPU/CPU thread. The implementation, however, must replicate them to each device that is involved in the computation, so that they can be used by the kernels. Hence, concurrent writes turn into a merge of updates after a parallel-for loop. Consistent with the OpenMP specification, it is the task of the programmer to avoid conflicting writes to the same data field.

JaMP splits *managed arrays* into chunks and partitions the data over all available devices. Fast devices automatically receive larger partitions to enhance load balancing. Kernels are co-located automatically, so that they access array elements locally and without costly communication. To improve the performance of typical stencil codes, array elements from neighboring chunks can be cached. In some cases, it is possible to detect the required overlap automatically. In other cases, JaMP provides a boundary clause to specify that size. Outside the parallel-for loop, read accesses or assignments are automatically transformed into getter or setter calls. These getters/setters copy the objects' data between the JVM and the C-structs on the accelerators.

Object members can be primitive data types (e.g., integers or floats), as well as objects or object arrays. However, it is important to know that, for performance reasons, unlike in Java's semantics, we do not store references to arrays or objects in fields, but rather directly inline them in the C-structs on the GPUs. Array members are likewise inlined into the object. With the resulting flat data layout, an object assignment in a parallel-for loop can turn into an efficient memory copy operation from the struct that represents the right-hand side object to the struct of the left-hand side object. The flat structs are also the key in achieving good data exchange performance when objects are shipped to/from the GPUs.

IV. Implementation

To execute a JaMP program on the cluster, the JaMP middleware must be started. The most important tasks of the middleware are the generation of the kernel codes and the data transfer between the different machines. Below, we describe implementation details of these tasks.

A. From parallel-for to kernels

The cluster's master node starts a single JVM supplied with the special JaMP classloader. All other cluster nodes start a C++ program that initializes the JaMP middleware only.

The classloader on the master node is in charge of the kernel generation and invocation. It replaces a parallel-for loop in the Java bytecode with a native call into the middleware to start the kernel. As mentioned before, to avoid the overhead of bytecode analysis at run-time, the JaMP compiler has already converted the parallel-for's body to a CKM internal representation that is now used to generate the CUDA/OpenCL kernels. As soon as the kernels are compiled, the middleware performs the following additional steps to execute the loop:

1) Use the CKM to allocate memory for all the shared objects that the kernels need.
2) Copy all the shared objects from the JVM to the accelerators. It is not necessary to copy the managed arrays, as their chunks are already placed on the correct devices. Subsection B covers the details.
3) Pass all managed arrays and shared objects as arguments to the kernels.
4) Invoke the kernels on every involved device.
5) Perform the merge step for shared objects, free memory, and copy shared objects back to the JVM. As managed arrays can still be accessed after the parallel-for loop, their memory must not be freed.

```
// Java class
public class Triangle {
   float area;
   FloatVector3D normal;
   FloatVector3D[] corners;
   Triangle() {
      corners = new FloatVector3D[3];
   }
}

// resulting C++ code:
struct FloatVector3D {
   float x, y, z;
}
struct Triangle {
   float area;
   struct FloatVector3D normal;
   struct FloatVector3D corners[3];
}
```

Figure 3. Structs generated for a JaMP class with different member types.

B. Data exchange between CPUs and GPUs

One of the main building blocks of our technique is the mapping between Java objects and C-structs and the data transfer between the two worlds. Note that the JaMP compiler pre-generates the structs and also stores them in the special bytecode attribute.

Let us look at the data structures first. Fig. 3 illustrates an example JaMP class and the data layout used in the corresponding C-structs. Primitive data type members are simply stored in the struct, whereas object references are changed as we inline the referenced object. Hence, generated structs do not contain pointers to other structs. This is crucial for the performance of JaMP, because it produces more efficient CUDA/OpenCL code, speeds up object transfer and removes time-consuming pointer indirection. This reduction of pointer indirections also leads to lower cache pressure.

To enable inlining, member arrays must be created with constant sizes. The outermost managed object arrays, which are passed to the kernel directly, are not bound by this restriction. Moreover, there may not be any recursion or cyclic dependencies in the members' data types.

To illustrate the problem with recursive object inlining, imagine a class **A** that has a member of type **B**. Let class **B** also have a member of type **A**. Since C-structs do not

support forward declarations, each type of a field in a struct must already be declared before the declaration of the struct that should contain the field. In the above cycle, **A** would require **B** to be declared first, while in contradiction **B** would require **A** to be the first struct in the source code. Only in the absence of cycles, the compiler can sort the structs topologically.

A flat struct data layout has a clear advantage in a distributed system. When another device requires a managed object-array element, the struct can be copied without further analysis of its contents, since there are no pointers to be respected inside the struct. Even the size of the object does not have to be computed, as it is a compile-time constant.

Another advantage of the flat struct data layout is that a merge of objects can be implemented in a generic way. Let us assume that the master node keeps a copy of a shared object's struct. After a parallel-for loop, each device's copy of that struct is compared to the original copy (in a byte-by-byte fashion), and all changes are collected to update the shared Java object in the JVM. This generic merge makes sure that, if any threads on the devices change one or more fields of a common shared object, all the modified bytes will be merged at the end of a parallel-for.

Let us now look at the implementation of the transfer of object data between the Java and the C world. For every Java class, the JaMP compiler generates getters or setters to copy the values of a Java object from the JVM to the according struct and back. Such a copy function transfers all fields of primitive type in one memcopy operation, and invokes other copy functions for the inlined objects. The combined copy operation speeds up the run-time communication of JaMP. Note that, depending on the location of the devices, copy operations may be remote through the SRPC layer.

As the code for getters and setters is also generated in advance by the compiler, the JaMP classloader can efficiently apply it in the generated code.

C. Limitations of the current prototype

As we recursively inline all objects into a managed array or shared object, only non-recursive type declarations are allowed for shared objects and managed arrays. This and other limitations, such as the currently missing support for exceptions or automatic memory management, do not impact the generality of our approach and are part of our future work. We have left out support for inheritance and interfaces because it simplifies the prototype's code generation, especially with respect to the structs' layout. OpenMP/JaMP synchronization constructs such as `critical` and `barrier` are not yet implemented in the current prototype.

V. Related work

JOMP [7] is another OpenMP-style programming framework for Java that, in contrast to JaMP, does not support clusters or GPUs.

Lee et al. [8] introduce a system that allows the use of GPUs from OpenMP, but it is also limited to a single GPU or machine. Their work on generating better GPU code from an OpenMP parallel-for loop is orthogonal to the techniques described here. In our previous work [9], [10], we described a system that already made use of multiple GPUs and machines, but did not allow objects or object arrays. This paper adds the efficient mapping of Java objects (and arrays thereof) to flat structs and their transparent placement in a heterogeneous cluster.

The manual handling of object arrays and other low-level details by a developer is also an unavoidable task if libraries like jCUDA [11] or jocl [12] are used. The Aparapi [13] open source project does not support object arrays, either.

We intentionally diverge from the Java standard inside a parallel-for loop, in contrast to other systems that attempt to keep the Java semantics intact in the GPU code. Rootbeer [14] and JaBEE [15] are two such systems. JaBEE [15] provides dynamic dispatch, encapsulation, and object creation on GPUs by performing bytecode translation. Their measurements show that dynamic dispatch costs performance. We actively avoid this by explicitly forbidding polymorphism. Rootbeer [14] likewise performs bytecode translation. In contrast to our approach, Rootbeer allows only a single GPU. JaBEE and Rootbeer both implement a Java reference with pointers whose dereferencing is costly. Our object inlining avoids this cost.

Traditional Distributed Shared Memory (DSM) systems use access checks to detect remote memory access. Access checks can either use a processor's MMU for instance in the TreadMarks DSM [16], or they are implemented in software, e.g. in Shasta [17]. Unfortunately, both approaches are unfeasible on a GPU.

Other works like PEPPHER [18] focus on the runtime system. For the future, it would be interesting to investigate whether it is possible to use other runtime systems like PEPPHER with the generated JaMP Kernels.

X10 [3] and Chapel [4] can make use of GPUs, but both require that a programmer uses them explicitly. For example, in X10, a GPU is abstracted into a 'place', while in JaMP, the use of GPUs is transparent and a programmer does neither have to know what accelerators there are nor their core counts.

A number of systems try to automatically inline objects. For example, Laud [19] discusses some required compiler analysis techniques. In our system, we avoid such complex analysis and let the programmer explicitly mark object arrays as `'managed'`.

VI. Measurements

All measurements were made on a cluster consisting of 5 machines with 2 GPUs each. Every machine is equipped with two 2.6 GHz Intel Xeon 5550 (Nehalem) processors. The GPUs are Nvidia Tesla M1060s each. First, we show the

Table I
RUN-TIME OF THE MICROBENCHMARKS IN SECONDS.

CUDA computation with flat structs	3.97
CUDA computation with object pointers	14.76
Serializing with flat structs	0.04
Serializing with object pointers	9.05

usefulness of our approach with two microbenchmarks. We also benchmarked three small applications, an *ArraySum*, a *VectorAddition*, and a *Raytracer* that generates a picture from a simple 3D scene. In this paper, we only show the CUDA results, as OpenCL behaves analogously. For comparisons we use Oracle Java 1.7 with the HotSpot JIT.

A. Microbenchmarks

Our main reason to use flat structs instead of objects with references to other objects is to avoid costly pointer indirections in both computations and data exchange. To demonstrate how much can be gained by this decision, we multiplied vectors of a million triangles in a CUDA kernel on a single GPU. In one version we used the flat structs from Fig. 3, in the other we retained pointers to triangles and vectors. The results in the upper two lines of Table I show that the version with flat structs is 3.7 times faster.

Minimizing the time of the data transfer between nodes is also crucial for a cluster. To ship an array of a million triangles in a flat data structure only takes 0.04s, whereas the serialization of Java's regular object references requires 9.1s. Hence, a factor of about 200 in data transfer overhead can be saved when it is no longer necessary to check for duplicate objects and to dereference pointers.

B. ArraySum

ArraySum works with a managed array of one million objects that contain two fields, a plain integer and an inlined array of 1024 integers. The body of the parallel-for loop (or the kernel) works on one of the 1 million objects, sums up all the values from the inlined array 32 times, and stores the result into its primitive integer. JaMP, using all CPUs and GPUs on one machine, takes 1.47s, see Fig. 4.

From one machine (with 2 GPUs) to 5 machines (10 GPUs), we see a speedup of 4.9, which is slightly below ideal due to a slowly growing communication and administration overhead.

C. Vector Addition

VectorAddition works with 4D int vector objects. There are two shared vector objects and a managed array of 16 million vector objects. The parallel-for loop adds the shared vectors 256 times to each of the vectors in the managed array. The performance results are similar to *ArraySum* in Fig. 4. For the *VectorAddition* benchmark, the execution time on one machine is about 7.0s instead of 1.6s on 5 machines (10 CUDA GPUs and 5 CPUs), resulting in a speedup of about 4.4.

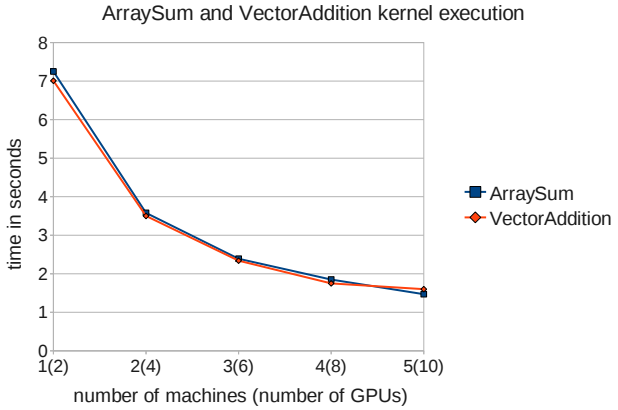

Figure 4. Run-time of ArraySum and VectorAddition on the cluster.

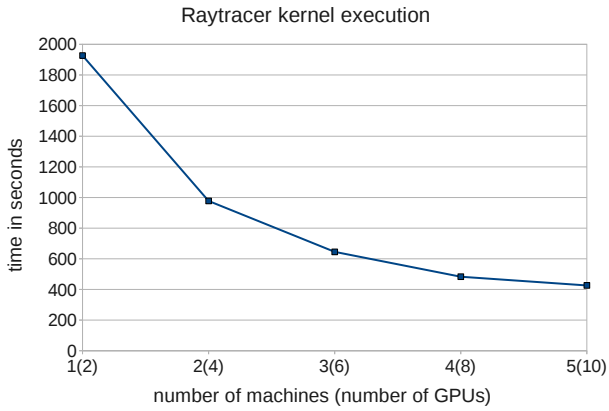

Figure 5. Run-time of the Raytracer example on the cluster.

D. Raytracer

Raytracers are computationally intensive applications that are straightforward to parallelize. They typically use objects and methods instead of the omnipresent arrays of primitive data types in typical stencil codes. Our implementation uses 17 Java classes, 8 managed arrays (mainly to provide temporary objects like *Intersections*, *Rays* and *Colors* for the kernels), and 5 shared objects (a *Scene* object for the *Triangles*, an *AreaLight*, the *Picture* with a *Color* array for the pixels, a *Camera* and a *LightingIntegrator*). The parallel-for loop uses 36 methods from these objects. Every iteration computes the color of a single pixel. Our benchmark uses 1024×1024 pixels, 32 samples per pixel, 144 light samples, only direct lighting and a Cornell box with 34 triangles as parameters. There is neither a bounding box hierarchy nor any other optimization of the ray-triangle intersection tests. This means that, for every ray, all triangles are tested

sequentially. The JaMP version on 5 machines (10 GPUs) is 4.5 times faster compared to one machine, see Fig. 5.

VII. Conclusion

We have presented a Java/OpenMP extension that allows transparent use of objects and object arrays distributed over a heterogeneous cluster equipped with many GPUs. Our system requires little effort (one JaMP directive for the parallel-for loop, and one for each managed array) to use a cluster with GPUs and MPI. It does not require to manually write the CUDA (or OpenCL) code of the kernel, the JNI communication between Java and C, and the distribution of the arrays and kernels across the network with MPI. Moreover, as the JaMP directives are considered as comments by a standard Java compiler, the resulting sequential code is employable on any JVM. To test the multi-threaded behavior of JaMP programs, the JaMP compiler can also generate standard Java code using Java threads (without the JaMP middleware). Once all code is tested, the full JaMP environment can be used to transparently run on a cluster.

Where other systems attempt to exactly implement Java's semantics in a parallel-for loop, we deliberately use object inlining for objects used in the loop. Object inlining is the key to performance. Measurements on small applications show that object inlining increases performance by a factor of 4.

Acknowledgements

This project has been supported in part by the Embedded Systems Institute (ESI) and the ESI-Anwendungszentrum, http://esi-anwendungszentrum.de/.

References

[1] L. Hochstein, F. Shull, and L. B. Reid, "The role of MPI in development time: a case study," in *Proc. Conf. on Supercomputing (SC'08)*, Piscataway, NJ, USA, November 2008, pp. 1–10.

[2] I. Christadler, G. Erbacci, and A. Simpson, "Performance and Productivity of New Programming Languages," in *Facing the Multicore-Challenge II*, ser. Lecture Notes in Computer Science, R. Keller, D. Kramer, and J.-P. Weiss, Eds. Springer Berlin Heidelberg, 2012, vol. 7174, pp. 24–35.

[3] 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. Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'05)*, San Diego, CA, Oct. 2005, pp. 519–538.

[4] 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, August 2007.

[5] S. Wienke, D. Plotnikov, D. Mey, C. Bischof, A. Hardjosuwito, C. Gorgels, and C. Brecher, "Simulation of bevel gear cutting with GPGPUs—performance and productivity," *Computer Science - Research and Development*, vol. 26, pp. 165–174, June 2011.

[6] "OpenMP Application Program Interface, Version 2.5," May 2005, `http://www.openmp.org/`.

[7] J. Bull and M. Kambites, "JOMP—an OpenMP-like Interface for Java," in *Proc. Conf. on Java Grande (JAVA'00)*, San Francisco, CA, June 2000, pp. 44–53.

[8] 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.

[9] M. Klemm, M. Bezold, R. Veldema, and M. Philippsen, "JaMP: An Implementation of OpenMP for a Java DSM," *Concurrency and Computation: Practice and Experience*, vol. 19, no. 18, pp. 2333–2352, December 2007.

[10] R. Veldema, T. Blaß, and M. Philippsen, "Enabling Multiple Accelerator Acceleration for Java/OpenMP," in *Proc. USENIX Workshop on Hot Topics in Parallelism (HotPar'11)*, Berkeley, CA, May 2011, pp. 1–6.

[11] "jCUDA," 2012, http://http://www.jcuda.org/.

[12] "jocl," 2012, http://www.jocl.org/.

[13] "Aparapi," 2012, http://code.google.com/p/aparapi/.

[14] J. R. Philip C. Pratt-Szeliga, "Rootbeer: Seamlessly using GPUs from Java," in *Proc. Conf. on High Performance Computing and Communication (HPCC'12)*, Liverpool, UK, June 2012, pp. 375–380.

[15] W. Zaremba, Y. Lin, and V. Grover, "JaBEE: framework for object-oriented Java bytecode compilation and execution on graphics processor units," in *Proc. Workshop on General Purpose Processing with Graphics Processing Units (GPGPU-5)*, London, UK, March 2012, pp. 74–83.

[16] P. Keleher, A. Cox, S. Dwarkadas, and W. Zwaenepoel, "TreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems," in *Proc. Winter 1994 Usenix Conf.*, San Francisco, CA, January 1994, pp. 115–131.

[17] D. Scales, K. Gharachorloo, and C. Thekkath, "Shasta: a low overhead, software-only approach for supporting fine-grain shared memory," in *Proc. Intl. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Cambridge, MA, October 1996, pp. 174–185.

[18] S. Benkner, S. Pllana, J. Traff, P. Tsigas, U. Dolinsky, C. Augonnet, B. Bachmayer, C. Kessler, D. Moloney, and V. Osipov, "PEPPHER: Efficient and Productive Usage of Hybrid Computing Systems," *Micro, IEEE*, vol. 31, no. 5, pp. 28–41, Sept.-Oct. 2011.

[19] P. Laud, "Analysis for Object Inlining in Java," in *Proc. Conf. Java Optimization Strategies for Embedded Systems (JOSES'01)*, Genoa, Italy, April 2001, pp. 1–8.