Cashmere: Heterogeneous Many-Core Computing

Pieter Hijma*, Ceriel J.H. Jacobs*, Rob V. van Nieuwpoort† and Henri E. Bal*
* *VU University Amsterdam, The Netherlands*
Email: {pieter,ceriel,bal}@cs.vu.nl
†*Netherlands eScience Center*
Email: R.vanNieuwpoort@esciencecenter.nl

***Abstract*—New generations of many-core hardware become available frequently and are typically attractive extensions for data-centers because of power-consumption and performance benefits. As a result, supercomputers and clusters are becoming heterogeneous and start to contain a variety of many-core devices. Obtaining performance from a homogeneous cluster-computer is already challenging, but achieving it from a heterogeneous cluster is even more demanding. Related work primarily focuses on homogeneous many-core clusters.**

In this paper we present Cashmere, a programming system for heterogeneous many-core clusters. Cashmere is a tight integration of two existing systems: Satin is a programming system that provides a divide-and-conquer programming model with automatic load-balancing and latency-hiding, while Many-Core Levels is a programming system that provides a powerful methodology to optimize computational kernels for varying types of many-core hardware. We evaluate our system with several classes of applications and show that Cashmere achieves high performance and good scalability. The efficiency of heterogeneous executions is comparable to the homogeneous runs and is >90% in three out of four applications.

***Keywords*-many-core; heterogeneous; cluster; divide-and-conquer**

I. Introduction

Many-core devices offer enormous potential in compute-power and can increase the performance of supercomputer and data-center applications significantly. Developments in the many-core field move fast: new generations of many-core hardware become available frequently and are fitted in data-centers because of performance and power consumption benefits. However, older-generation accelerators may still be powerful for some applications. For instance, in our DAS-4 cluster, we have older-generation Fermi GTX480 GPUs that for some applications are as fast as the newer generation Kepler GTX680 GPUs [1].

The result is that supercomputers and data-centers will more and more contain multiple types of many-core hardware. As an example, our DAS-4 cluster [2] has a wide variety of many-core hardware, multiple generations of NVIDIA GPUs, AMD GPUs, and Intel Xeon Phi's. As another example, Table I shows several TOP500 supercomputers (as of November 2014) that contain more than one type of many-core device [3].

Extracting performance from many-core clusters is difficult in general, but heterogeneity makes it even more challenging. There is a variety of problems that need to be solved to obtain high-performance:

1) Since execution times are likely to vary among many-core devices, some form of load-balancing is needed.
2) Load-balancing is especially challenging for heterogeneous many-core clusters, because it requires communication between nodes. However, because many-cores are so fast, the network speed is relatively slow compared to clusters without many-cores, resulting in a skewed computation/communication ratio.
3) The computational kernels that will run on the many-core devices have to be identified, written, and optimized for each type of hardware.
4) The run-time system should know the type and number of available many-core devices in each node and map the right kernels to the devices.
5) The application needs logic to drive the execution on these many-core devices.

Related work focuses mostly on extracting high-performance from *homogeneous* many-core clusters, an already difficult task. Often, the overall solution is based on MPI combined with a programming language for many-core devices, such as OpenCL [4] or CUDA [5]. However, because execution times vary among types of many-core hardware, a solution based on MPI with its rigid communication patterns is not the ideal solution.

In this paper, we present Cashmere, a programming system that focuses on obtaining performance from *heterogeneous* many-core clusters. Cashmere is a tight integration of two existing systems: Satin, a programming system that provides a divide-and-conquer programming model with load-balancing and latency-hiding [6] and Many-Core Levels (MCL) providing a powerful methodology to optimize computational kernels for varying types of many-core hardware [7]. Satin addresses problems 1) and 2) described above, MCL addresses problem 3), and Cashmere integrates the two systems providing solutions for problems 4) and 5) and extends Satin's load balancing for multiple many-core devices per node. We named our system Cashmere because it offers parallelism in the form of fine-grained threads.

Our contributions are the following:

- We seamlessly integrate coarse-grained divide-and-

Table I
TOP500 SUPERCOMPUTERS WITH HETEROGENOUS MANY-CORE DEVICES.

name	institute	ranking	configuration
Quartetto	Kyushu University	49	K20, K20X, Xeon Phi 5110P
Lomonosov	Moscow State University	58	2070, PowerXCell 8i
HYDRA	Max-Planck-Gesellschaft MPI/IPP	77	K20X, Xeon Phi
SuperMIC	Louisiana State University	88	Xeon Phi 7110P, K20X
Palmetto2	Clemson University	89	K20m, M2075, M2070
Armstrong	Navy DSRC	103	Xeon Phi 5120D, K40
Loewe-CSC	Universitaet Frankfurt	179	HD5870, FirePro S10000
Inspur TS10000	Shanghai Jiaotong University	310	K20m, Xeon Phi 5110P
Tsubame 2.5	Tokyo Institute of Technology	392	K20X, S1070, S2070
El Gato	University of Arizona	465	K20, K20X, Xeon Phi 5110P

conquer parallelism with multiple fine-grained many-core parallelism levels for a variety of many-core devices with minimal changes to the original Satin programming model (Sec. II).
- We describe several optimizations that are necessary to obtain high-performance (Sec. III).
- We demonstrate that our stepwise-refinement for performance methodology enables us to develop large amounts of optimized many-core kernels. This scalable development is a prerequisite for achieving the heterogeneity that Cashmere targets (Sec. IV and V).
- We evaluate Cashmere with several classes of applications and show that we can achieve good scalability and performance despite the skewed communication and computation ratio (Sec. IV and V). Our heterogeneous runs are comparable to homogeneous runs and in three of four applications we achieve >90% efficiency with optimized kernels.

After an overview of related work (Sec. VI) we present our conclusions (Sec. VII).

II. CASHMERE PROGRAMMING MODEL

Cashmere is a system for programming high-performance applications for clusters with many-core accelerators and forms a tight integration of two existing programming systems: Satin [6] and MCL [7]. Section II-A gives a brief overview of Satin and its programming model, Sec. II-B describes MCL, and Sec. II-C discusses how Satin and MCL interact and form the Cashmere programming model.

A. Satin

Satin [6] is a programming system that targets grids or clouds of clusters. It is inspired by Cilk [8] that targets multi-core processors. Similar to Cilk, Satin has a divide-and-conquer programming model that allows programmers to express computation in a hierarchical manner. Satin achieves very good scalability by mapping this computation to the hierarchical structure of grids or clouds of clusters.

The key features of Satin that we use are:

- **load-balancing** Satin uses random work-stealing to achieve load-balancing.

```
1 spawnable f(a) {
2   if (small_enough_for_leaf(a)) {
3     return do_leaf_computation(a)
4   }
5
6   r1 = f(make_smaller(a)) //asynchronous
7   r2 = f(make_smaller(a)) //asynchronous
8   sync
9
10  return combine(r1, r2)
11 }
```

Figure 1. Skeleton of a Satin program.

- **latency hiding** Overlap slow communication with computation.
- **fault tolerance** Satin recovers from nodes that are no longer responding.
- **shared objects** Shared objects make the divide-and-conquer programming model less restrictive by allowing programmers to use a custom consistency model.

Figure 1 shows a basic skeleton of a Satin program in pseudo-code. Line 1 shows a recursive function f() that is declared to be spawnable, which means that a function call will execute asynchronously. Lines 6 and 7 show recursive calls to f(). The Satin system creates a child job for each call that will be asynchronously computed on a compute node that Satin has available. This may be the same node, another node in the cluster, or a node on another cluster. The results of the two calls are stored temporarily in r1 and r2 but are not available until the sync statement has finished. The function blocks on the sync statement until all previous child jobs have finished. After the sync statement, the function can return the result based on r1 and r2.

The above mechanism creates (possibly) recursive jobs that are spread among the compute nodes of clusters. Satin achieves good scalability since all nodes steal from each other using random work-stealing. At some point, the jobs are small enough to perform the actual computation. Lines 2 to 4 show the stop-condition based on an application defined parameter that decides to perform the leaf computation. Section II-C will show how we extend this model with MCL.

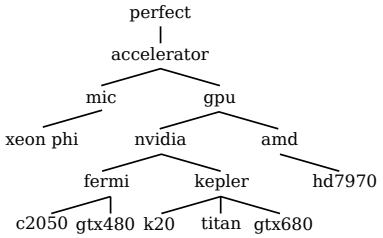

Figure 2. Hierarchy of hardware descriptions.

B. MCL

Many-Core Levels is a programming system that allows programmers to write computational kernels for many-core hardware. The programming system is designed such that programmers can choose their own abstraction-level for their program to trade-off performance, portability and ease of programming.

MCL offers abstraction levels by providing a library of hardware descriptions that are organized in a hierarchy. Figure 2 shows the hierarchy of hardware descriptions used for Cashmere. Each child hardware description specifies more details about the many-core hardware than its parent. At the root is hardware description *perfect* that describes idealized hardware with unlimited compute units and 1 cycle latency for memory, providing a high level of abstraction.

MCL contains two languages: the hardware description language HDL allows one to define many-core hardware and the programming language MCPL is used to express the computational kernels. The languages are designed such that the compiler and programmers understand how the computation is mapped to the hardware. The compiler leverages this knowledge to give performance feedback.

As such, the system supports a methodology that we call "stepwise-refinement for performance": Programmers choose an initial hardware description, receive feedback from the compiler, and modify the program manually until there is no feedback left. Subsequently, the compiler can translate the program automatically to a lower-level hardware description. The optimization process starts again, but on this level the compiler can give more detailed feedback because it has more hardware knowledge.

Figure 3 shows a matrix multiplication kernel written in MCPL for hardware description *perfect*. MCPL is a C-like language with multi-dimensional arrays that keep track of their sizes. The keyword perfect on line 1 indicates that the kernel is written for hardware description *perfect*. The two foreach statements on lines 4 and 5 express parallelism of n and m threads. The keyword threads refers to an identifier defined in the hardware description and gives the compiler information on how the expressed parallelism is mapped to the hardware.

```
1 perfect void matmul(int n, int m, int p
2     float[n,m] c,
3     float[n,p] a, float[p,m] b) {
4   foreach (int i in n threads) {
5     foreach (int j in m threads) {
6       float sum = 0.0;
7       for (int k = 0; k < p; k++) {
8         sum += a[i,k] * b[k,j];
9       }
10      c[i,j] += sum;
11 } } }
```

Figure 3. Matrix multiplication written in MCPL.

C. Cashmere programming model

Cashmere is targeted at leveraging the fine-grained parallelism that many-core hardware offers on a large scale, typically clusters with compute nodes that contain a variety of many-core hardware such as GPUs or Intel's Xeon Phi.

Adding many-cores to the original Satin programming model provides additional levels of parallelism:

1) *parallelism on a many-core device* A many-core device itself exposes many levels of parallelism: instruction-level parallelism, task parallelism, and SIMD parallelism.
2) *multiple many-core devices per node* A compute node in a cluster can contain multiple (heterogeneous) many-core devices.
3) *overlap in communication and computation* Typically, many-cores are connected through a PCI Express bus. Many-core devices can overlap communication between host and device with computation.

Cashmere aims to exploit these levels of parallelism with minimal changes to the original Satin programming model and leveraging MCL for writing the computational kernels.

1) parallelism on a many-core device: Expressing the parallelism on the device is completely handled by MCL. Programmers write a kernel in MCL's programming language MCPL targeting a hardware description from the library. The MCL compiler will generate code for each of the leaf hardware descriptions and glue-code for Cashmere. This results in a minimal and convenient way to call MCL code from within Cashmere while maintaining Satin's fault-tolerance. Figure 4 shows a possible leaf computation. In this figure, the MCL kernel needs parameters a and b (lines 1 and 5). On line 3 in the try/catch clause, we retrieve the MCL kernel from Cashmere. From the kernel we create a launch kl on line 4 that we launch with the MCL front-end with parameters a and b. The MCL front-end makes sure that all necessary data is copied to the many-core device, it selects the appropriate kernel(s) for the devices available on the node, executes the kernel, and copies the data back. In case something goes wrong with the kernel execution, the system raises an exception which will then start the leaf computation on the CPU (line 7).

Figure 4 shows the basic scheme, but more advanced schemes are possible:

```
1 leaf(a, b)
2   try {
3     Kernel kernel = Cashmere.getKernel()
4     KernelLaunch kl = kernel.createLaunch()
5     MCL.launch(kl, a, b)
6   catch (exception) {
7     leafCPU(a, b)
8 } }
```

Figure 4. Calling an MCL kernel in Cashmere.

- **multiple kernels** The above scheme works for an application with only one kernel. Cashmere will automatically find this kernel and load it. If there are more kernels, the Cashmere.getKernel() function should have a string parameter that identifies the kernel to be loaded.
- **multiple kernel-launches** It is possible to launch the kernel multiple times in succession. For example, it is possible to put a loop around lines 4 and 5.
- **device copies** If there are multiple kernel launches, it may be unnecessary to transfer all the parameters to the many-core device each time. Cashmere offers the functions Kernel.getDevice() and Device.copy() to copy data to and from the many-core device for multiple kernel launches.

2) multiple many-core devices per node: If there are multiple many-core devices on a node, the standard Satin programming model and calling MCL as explained in the above paragraph will not lead to parallel execution of MCL kernels on both devices, because a call to MCL.launch() is blocking. We solved this problem with a minimal change to the divide-and-conquer programming model. Since the spawnable functions already express parallelism, we reuse this to express parallelism for the many-core devices. Figure 5 shows a skeleton of a typical Cashmere program. Compared to the Satin skeleton in Fig. 1, lines 5 to 7 have been added. If on line 5 the job is small enough, then the function Cashmere.enableManyCore() disables generating jobs for the compute nodes in the cluster. Instead, it continues to create jobs using the same mechanism of spawnable functions and sync, only now for the many-core devices on a node. If in turn the jobs generated for the many-core devices are small enough to execute the leaf computation, governed by the stop-condition on line 2, then the leaf computation running the MCL kernels will be started. In conclusion, by adding one library function, the divide-and-conquer model of Cashmere expresses parallelism among nodes and among many-core devices on a node.

3) overlap in communication and computation: The above mechanism also overlaps computation and communication to the many-core device over the PCI Express bus. If a node has multiple jobs available, Cashmere can launch kernels for one job and copy data for another. Cashmere automatically manages the available memory on a device.

In summary, the Cashmere programming model is similar

```
1 spawnable f(a) {
2   if (small_enough_for_leaf(a)) {
3     return do_leaf_computation(a)
4   }
5   else if (small_enough_for_many_core(a)) {
6     Cashmere.enableManyCore()
7   }
8
9   r1 = f(make_smaller(a)) //asynchronous
10  r2 = f(make_smaller(a)) //asynchronous
11  sync
12
13  return combine(r1, r2)
14 }
```

Figure 5. Skeleton of a Cashmere program.

to the original Satin programming model. It is extended with a library call to express parallelism between many-core jobs and it provides a simple front-end to call MCL kernels. MCL is used to write the kernel code and is extended to generate glue code for Cashmere.

III. Implementation

This section describes the Cashmere implementation. Section III-A explains the role of MCL in Cashmere while Sec. III-B describes how the MCL kernels fit in the divide-and-conquer system.

A. MCL

To write kernels for multiple many-core devices, Cashmere makes use of several capabilities of MCL:

translation between abstraction-levels: Hardware descriptions consist of definitions for the physical device and for the programming abstractions that define how code is mapped to the physical device. For example, in Fig. 3 the keyword threads on lines 4 and 5 refers to a programming abstraction defined in the hardware description that tells how the threads are mapped to the compute units. MCL can automatically translate kernels written for the programming abstractions of hardware description *x* to the programming abstractions of a child level *y*. Since each lower-level hardware description contains more detailed information, the mapping between programming abstraction and physical device becomes more precise. During this translation process the compiler does not apply optimizations.

generating OpenCL code: For each leaf node in the hierarchy, there is a configuration file that tells the compiler how the programming abstractions from the hardware description map onto the OpenCL constructs. This means, together with translation between abstraction levels, that MCL can generate code from each abstraction level.

Generating Cashmere code: Applying the stepwise-refinement methodology leads to multiple files with different versions of the same kernel. For instance, given a kernel written on level *perfect*, suppose programmers know the AMD HD7970 GPU well and choose to apply optimizations on level *gpu*, *amd*, and *hd7970*. This leads to four different

files: a file with a kernel on level *perfect* and files on levels *gpu*, *amd*, and *hd7970*. The programmers can select these files and generate Cashmere code for these devices.

MCL will generate OpenCL code for each of the seven leaf nodes in Fig. 2 and automatically chooses the most specific kernel version for a device. This means that in the above situation, the Xeon Phi has a kernel on level *perfect*, all NVIDIA GPUs have kernels on level *gpu* and the HD7970 GPU has a kernel on level *hd7970*.

Together with the OpenCL code, MCL automatically generates glue code that calls the kernels with the right configuration for OpenCL's work-groups and work-items (parameters that determine the available parallelism) for inclusion in the divide-and-conquer framework of Cashmere.

This is important because the different devices have different granularity needs. For example, the Xeon Phi needs more coarse-grained parallelism than a GPU. MCL determines the work-group and work-item configuration based on the kernel parameters and its hardware-descriptions.

B. Cashmere

This section explains how Cashmere runs an application.

On initialization: In the initialization phase of an application, Cashmere assigns one node to be the master; the others become slaves. The kernels for the many-core devices may depend on run-time information that only the master node has. Therefore, the master broadcasts this run-time information to each slave. On each node, Cashmere retrieves which devices are available and after receiving the run-time information, compiles the most specific kernels for its compute devices. If there is a device on a compute node that is not available in the hierarchy of hardware descriptions, Cashmere suggests to add a hardware description for this device, so that it can compile a kernel for this device.

spawning jobs to other nodes: If the application encounters a spawnable function, Cashmere generates jobs that can be stolen by other nodes. The master is the first that generates jobs that other nodes can steal. As soon as a node has jobs, each node in the cluster can randomly steal from other nodes which contributes to scalability and load-balancing. Stealing a job encompasses transferring the input data to the requesting node, execution of the job on the requesting node (possibly generating new jobs that can be stolen) and transferring back the output data. This all happens automatically.

spawning jobs to the many-core devices: If the application encounters $\mathrm{Cashmere.enableManyCore()}$ Cashmere switches to a new mode. On encounter of a spawnable function, Cashmere no longer generates jobs that other compute nodes can steal, but instead it creates a thread that executes the spawnable function. The main thread will continue executing, possibly creating new threads for spawnable functions and will block on the sync statement until all threads have finished.

These threads can either generate more threads or encounter a call to an MCL kernel. In the last case, the input data for the kernel is scheduled to be copied to the device, the kernel is scheduled to run after the copies have completed, and the data transfer back to the device is scheduled to be run after the kernel execution. Because multiple threads are scheduling transfers and kernels, the data transfers can be completely overlapped with kernel executions except for the first and last data transfers.

Cashmere automatically load-balances the jobs scheduled to run on the many-core devices of a compute node. Initially, Cashmere uses a heuristic based on a static table of relative many-core device speeds to schedule the first jobs. For example, the table states that a K20 GPU has speed 40 and a GTX480 speed 20. When these jobs have completed, we know the execution time for each kernel for a specific device. Based on this time Cashmere submits the jobs to the different queues for each device trying to minimize the overall execution time for all jobs. For example, if the queue for a K20 has 3 jobs with an execution time of 100ms and the queue for the GTX480 has a queue with one job of 125ms, then Cashmere submits the job to the GTX480 queue because the execution time of this scenario is less: $min(scenario1, scenario2)$ where $scenario1 = max(4 * 100ms, 1 * 125ms)$ and $scenario2 = max(3 * 100ms, 2 * 125ms)$. This is possible because leaf jobs in a divide-and-conquer application typically have the same size.

IV. Methodology

In this section we describe our methodology to show that Cashmere obtains high performance on heterogeneous many-core clusters. Our test-bed is the main DAS-4 cluster [2], consisting of 74 dual Xeon E-5620 quad-core nodes that communicate with a QDR Infiniband interconnect. We evaluated Cashmere with the seven many-core devices available on this cluster:

- 22 NVIDIA GTX480 GPUs
- 8 NVIDIA K20 GPUs
- 2 Intel Xeon Phi (each fitted in a K20 node)
- 2 NVIDIA C2050 GPUs
- 1 NVIDIA Titan GPU
- 1 NVIDIA GTX680 GPU
- 1 AMD HD7970 GPU

We use 4 applications to evaluate Cashmere. Each application has its own characteristics and represents a class of applications. Table II shows an overview of how we classify each application to evaluate Cashmere.

Raytracer: This application is based on smallpt [9], and its GPU port SmallptGPU [10]. It is a path tracing raytracer that leads to very realistic images given that each pixel is computed with many random samples from each object in the scene. This is an interesting application for Cashmere because of two reasons: First, the application is highly compute intensive. The amount of data that is

Table II
THE CLASSES OF APPLICATIONS THAT WE USE TO EVALUATE CASHMERE.

application	type	computation	communication
raytracer	irregular	heavy	light
matmul	regular	heavy	heavy
k-means	iterative	moderate	light
n-body	iterative	heavy	moderate

processed is $\mathcal{O}(no)$ where n is the number of pixels and o the number of objects in a scene. The computation is $\mathcal{O}(nods)$ where d is the depth (the number of times a ray is bounced off an object) and s is the number of random samples. The number of samples determines the quality of the picture. Raytracer is also interesting because, although the application is compute-intensive, the application is highly irregular because of the random samples. The application has much control-flow based on randomness, making it difficult to optimize. Raytracer represents the class of highly parallel and compute-intensive irregular applications.

Matrix Multiplication: Matrix multiplication multiplies two dense matrices of single-precision floating-point numbers. For multiplying two $n \times n$ square matrices, the amount of data that is processed in $3n^2$. The amount of computation is about $2n^3$, which means that we have a factor n more computation than communication.

This application is interesting for evaluating Cashmere because although it is a compute-intensive application, the application is highly regular, making it relatively easy to optimize. However, this then results in relatively high communication costs making this application difficult to scale. This application represents the class of regular, compute- and communication-intensive applications.

K-means: K-means clusters a set of n d-dimensional data-points in k clusters. Given an initial set of d-dimensional centroids that represent the k clusters, k-means assigns the n data-points based on the distance to the centroid of the cluster. Based on the set of data-points belonging to one of the centroids, a new value for the centroid is computed. This process is repeated until there are no changes.

This application is interesting for evaluating Cashmere because it is an iterative algorithm that needs to update k values after each iteration and distribute these back to the compute nodes. A single iteration is compute-intensive and spawns the jobs over the nodes. For one iteration, communication is $\mathcal{O}(k)$ and computation is $\mathcal{O}(kn + k)$.

This application represents the class of iterative applications with minimal (constant) communication between the iterations.

N-body simulation: This application simulates the forces between n bodies over time. As K-means, it is an iterative application but has a different complexity for computation and communication. Each iteration, the effect of each body on each other body has to be computed, which makes the computation $\mathcal{O}(n^2)$. After each iteration, the positions and accelerations have to be updated for each body, making the communication $\mathcal{O}(n)$.

N-body represents the class of iterative applications with intensive communication and is interesting for evaluating Cashmere because of its communication pattern (all-to-all for each compute node).

We use MCL to write kernels for each application. First, we write a kernel on level *perfect* and generate code for each of the 7 leaf hardware descriptions in Fig. 2. This is a kernel written on a high level and we consider this the *unoptimized* kernel. For each application we also optimize for each device. We consider this the *optimized* version. In [7] we compared the performance of MCL applications against that of hand-optimized applications from the literature, showing that MCL performance overall is in line with other published results.

We then perform the following scalability studies on one type of hardware:

- **Satin** These measurements show the original performance of Satin and how well it scales. The results from these measurements help to put in perspective the performance that we obtain with many-core hardware and the scalability that we achieve.
- **Cashmere with non-optimized kernels** These measurements show the scalability and performance difference between Satin and Cashmere with minimal effort because the kernels are written on a high level.
- **Cashmere with optimized kernels** These measurements show the performance of Cashmere when the computational time is reduced several factors.

Finally, we evaluate heterogeneous runs using various types of hardware and compute the efficiency by dividing the measured performance by the maximum attainable performance. We determine this by summing the measured performance for one node for each node in the configuration. We compare the efficiency to the efficiency of the homogeneous execution. We use the optimized kernels to evaluate the heterogeneous runs. For each study we are strong scaling the problem.

V. EVALUATION

The following subsection discusses the kernel performance differences between the optimized and non-optimized versions. Section V-B shows the scalability studies and the absolute performance difference between the applications. Finally, Sec. V-C presents our findings on heterogeneous runs.

A. Kernel performance

This section shows the effect of the "stepwise-refinement for performance" methodology applied to the computational kernels. Since the hardware descriptions are organized in

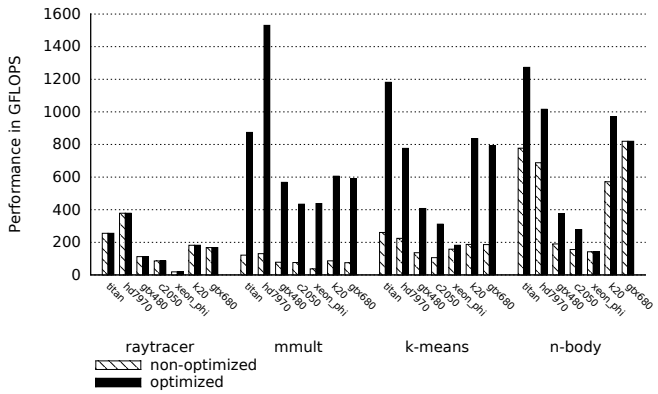

Figure 6. The performance of the kernels for the applications for the unoptimized version and the optimized version.

a hierarchy, optimizing the kernels becomes scalable. For example, optimizations that have been applied on level *gpu* are used for both NVIDIA and AMD GPUs, so these optimizations are used for four different devices.

Fig. 6 shows the kernel performance. The performance numbers are based on the timings of kernel execution alone without any overhead such as copying data to the device. It is clear that optimizing has a drastic effect on the kernel performance for most devices except for Raytracer. This can be explained by the irregularity of the kernel. Raytracer has much control-flow overhead and since the control-flow is based on randomness, threads often diverge, which has severe performance penalties. To obtain better performance from the raytracer would mean a different algorithm, something MCL cannot suggest. Raytracing is known to be challenging on many-cores and Xiang et al. discuss hardware solutions for this kind of kernels [11].

B. Scalability

This section evaluates whether Cashmere is able to obtain similar scalability results as Satin, which is our aim. We also compare the absolute performance difference between Satin and Cashmere. To our surprise, Satin scales worse than Cashmere in most of the cases. We found that there are two factors that contribute to this reduced scalability. Firstly, Satin has more overhead in job creation because it needs to create 8 times more jobs to keep one node busy. This can be explained by the difference in programming models: In the Satin programming model, a leaf computation is single-threaded, whereas one node has two quad-core processors, which means that to keep one node busy, Satin has to run 8 jobs in parallel. In contrast, a leaf computation in Cashmere already exposes parallelism for the many-core device. Hence, Cashmere does not have to create as many jobs as Satin. The second factor that contributes to worse scaling is that since all cores on the CPUs are fully occupied with computation, communication and load-balancing tasks suffer from the lack of available compute-power.

Table III
PERFORMANCE OF THE HETEROGENEOUS EXECUTIONS.

application	performance (GFLOPS)	configuration
raytracer	1883	10 gtx480, 2 c2050, 1 gtx680, 1 titan, 1 hd7970
matmul	3927	10 gtx480, 2 c2050, 1 gtx680, 1 titan, 1 hd7970
k-means	10644	10 gtx480, 2 c2050, 1 gtx680, 1 titan, 1 hd7970, 7 k20, 1 xeon_phi
n-body	13517	10 gtx480, 2 c2050, 1 gtx680, 1 titan, 1 hd7970, 7 k20, 2 xeon_phi

1) Raytracer: Figure 7 shows that Raytracer scales better with Cashmere than with Satin. This is a good achievement as the absolute performance of Cashmere is an order of magnitude higher as shown in Fig. 8. Since the optimized and non-optimized kernels are similar in performance, the two Cashmere versions overlap. We performed our measurements on the Cornell scene [9], [10] with a resolution of 16384×8192 with 500 random samples.

2) Matrix Multiplication: Figure 9 shows the performance results of multiplying 2 32768×32768 single-precision floating point matrices. We can see that Matrix Multiplication does not scale very well, also for Satin. The graph makes clear that the scalability suffers from the relatively slower networking speed when the kernel is optimized. However, Fig. 10 shows that there is still a factor four absolute performance difference between optimized and non-optimized versions.

3) K-means: Figure 11 shows that K-means scales well, even for the optimized version and better than Satin. Figure 12 shows the absolute performance of the three versions. We computed 4096 clusters out of 268 million points with 4 features in three iterations.

4) N-body: Figure 13 and 14 show that N-body has similar results as K-means despite higher communication costs. We simulated two iterations of 2 million bodies.

C. Heterogeneity

In Table III we show the performance of the four applications with 2 configurations dependent on the availability of nodes on the cluster. For the K-means and N-body experiments all 7 types of hardware were available simultaneously. We used the optimized kernels for our measurements. Fig. 15 shows the efficiency of the applications compared to the combined performance of one-node execution for each type hardware (i.e., the sum of $10\times$ #GFLOPS of a one-node execution on the GTX480, $2\times$ #GFLOPS of the one-node execution on the C2050, etc.). We compared this to the efficiency of the homogeneous executions for 16 GTX480 nodes from Sec. V-B. We conclude that the efficiency for heterogeneous runs is similar to the homogeneous runs.

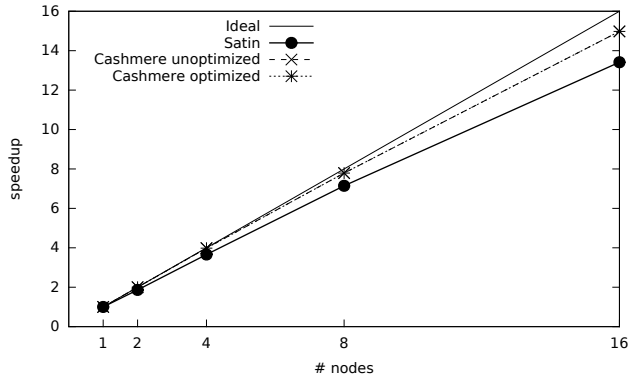

Figure 7. Scalability of Raytracer up to 16 GTX480 GPUs.

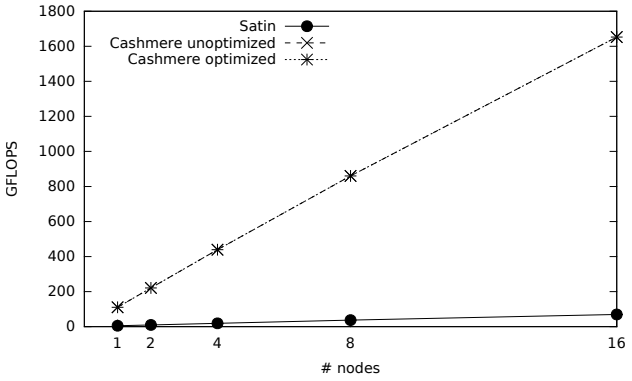

Figure 8. Absolute performance of Raytracer up to 16 GTX480 GPUs.

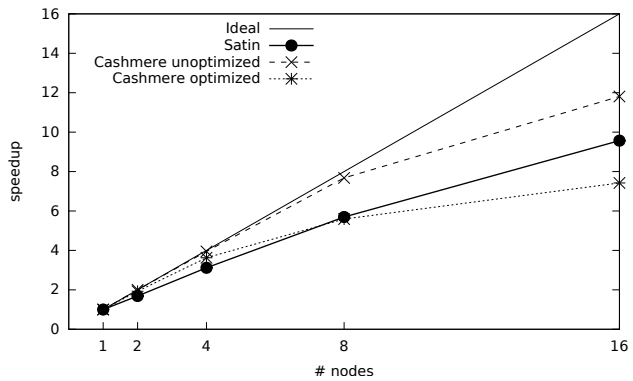

Figure 9. Scalability of Matrix Multiplication up to 16 GTX480 GPUs.

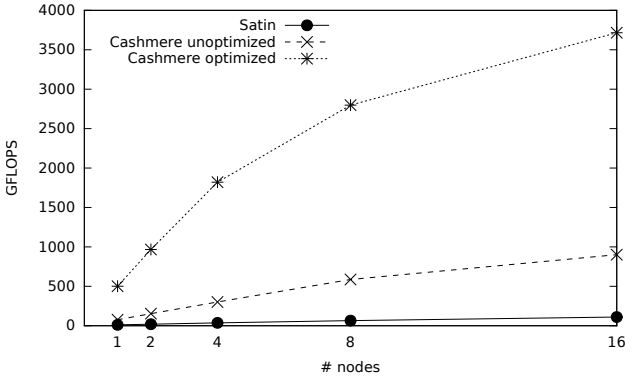

Figure 10. Absolute performance of Matrix Multiplication up to 16 GTX480 GPUs.

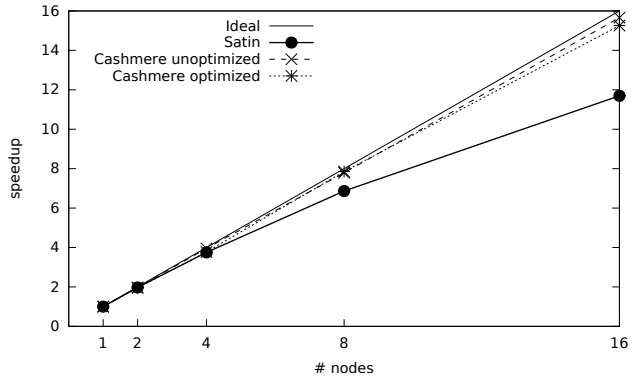

Figure 11. Scalability of K-means up to 16 GTX480 GPUs.

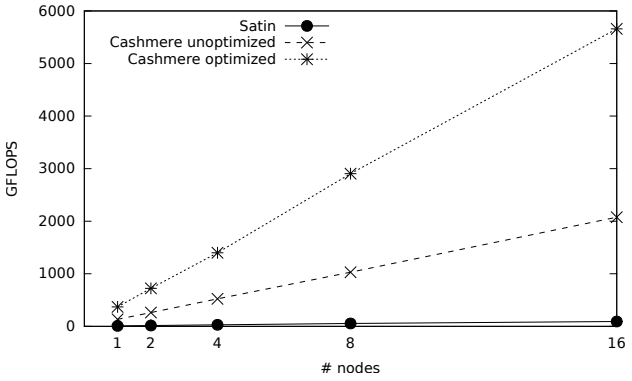

Figure 12. Absolute performance of K-means up to 16 GTX480 GPUs.

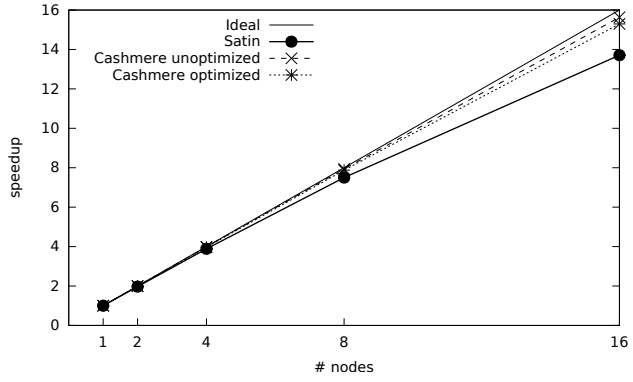

Figure 13. Scalability of N-body up to 16 GTX480 GPUs.

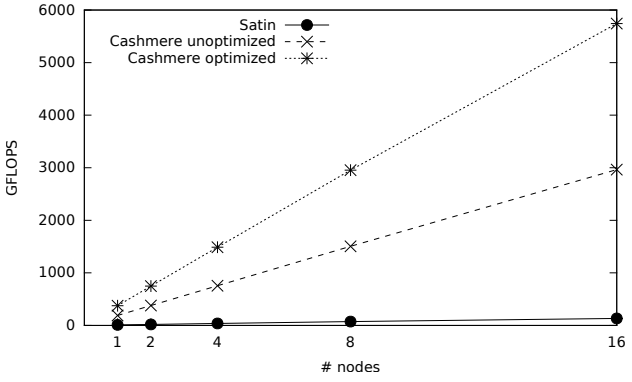

Figure 14. Absolute performance of N-body up to 16 GTX480 GPUs.

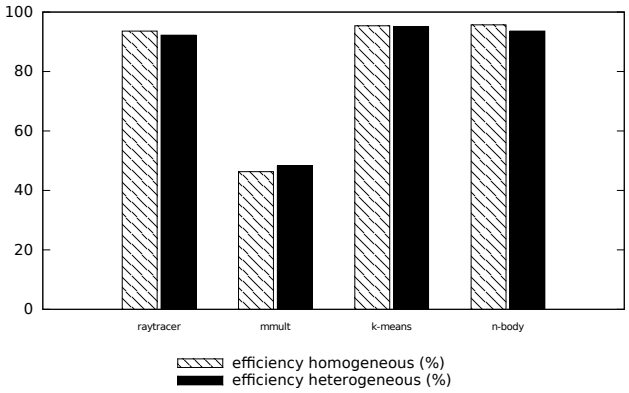

Figure 15. Efficiency of heterogeneous executions.

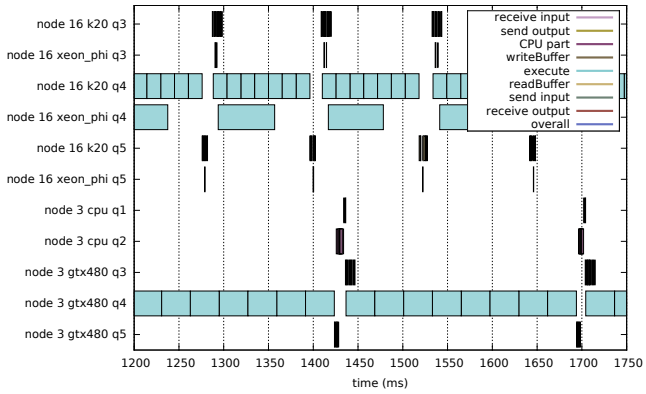

Figure 16. Zoomed-in view of the Gantt chart of heterogeneous K-means execution.

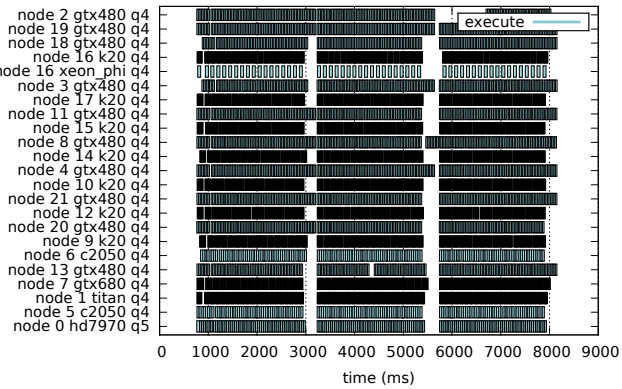

Figure 17. Gantt chart of heterogeneous K-means execution.

These results are especially impressive considering that in Matrix Multiplication the performance between the kernel versions varies widely (Fig. 6), thus showing that Cashmere prevents load-imbalance very well. The results for K-means and N-body are noteworthy as well because the efficiency is about the same as the homogeneous execution while using one third more nodes, achieving twice the performance with the heterogeneous runs, and after each iteration communication between all nodes is necessary.

Finally, to gain insight in these excellent results, we show a Gantt chart of the K-means run. Since there is so much parallelism, it is difficult to show all details. We therefore show a zoomed-in version of the Gantt chart (Fig. 16) where we can see two nodes, one with a GTX480 GPU and one with a Xeon Phi and a K20 GPU. The y-axis shows different queues denoted with 'qn'. Each queue contains activities that can be overlapped with activities in other queues. The narrow bars are for CPU tasks, sending data from and to the device and to other nodes. The wider bars in q4 are kernel executions. The chart shows parallel execution of a Xeon Phi kernel, overlapped with (faster) K20 kernels and GTX480 kernels on node 3.

Figure 16 also shows our load balancing algorithm at work explained in Sec. III-B. Node 16 executes sets of 8 jobs and after each set synchronization is required. Our load balancing algorithm schedules 1 job on the Xeon Phi and 7 on the K20 which is the fastest configuration. Since the Xeon Phi is about 4 times slower than the K20 (see Fig. 6), one job more on the Xeon Phi would lead to a longer overall execution time and not scheduling a job on the Xeon Phi would also be longer.

Figure 17 shows the zoomed-out version of the Gantt chart in which we left out all activities except kernel executions. It shows that this kind of execution can be maintained each iteration, thus showing Cashmere's effectiveness.

VI. Related Work

In this section we focus on programming systems that – similar to Cashmere – aim to simplify many-core computing on clusters and supercomputers.

MapReduce is a framework that allows programmers to express computations in terms of map and reduce functions [12]. Several frameworks target many-core clusters. GPMR is a Map-Reduce framework for GPU clusters [13]. It fully relies on CUDA which makes the framework not heterogeneous. Their scalability study does not take into account copying data back and forth between compute-nodes which is essential in practice.

HadoopCL [14] extends Hadoop [15] with OpenCL to make the computational power of many-cores available to Hadoop jobs. It uses APARAPI [16] to translate limited Java code to OpenCL. Cashmere offers similar functionality in a library to make use of MCL. HadoopCL performance is only compared to original Hadoop and obtains a speedup of 5 on 10 nodes with GPUs against 2 nodes of Hadoop without GPUs on the K-means application. Cashmere is an order of magnitude faster when compared to original Satin (a speedup of 186 on 8 GPU nodes compared to 2 Satin nodes).

Glasswing [17] is a MapReduce framework fully written in OpenCL and C++ and has a significant performance

benefit over the previous two frameworks. Glasswing is set up on top of the Hadoop filesystem and gains performance by overlapping computation with I/O in several deep pipelines. Glasswing supports multiple architectures but was not evaluated with multiple different architectures at the same time. Glasswing also performs a $32k \times 32k$ Matrix Multiplication on the DAS-4 and obtains a performance of 2082 GFLOPS on 16 GTX480 GPUs while we obtain 3716 GFLOPS with the optimized version. However, Glasswing supports out-of-core data which Cashmere does not support yet.

OmpSs [18] combines OpenMP pragmas with StarSs [19] pragmas to program clusters of GPUs. It offers a sequential programming model in which programmers can annotate parallel regions with directives to indicate the data-dependencies with in, out, and inout statements between tasks and on what kind of device a task should run.

Planas et al. [20] extended the system with an adaptive scheduler that can choose multiple versions of compute-kernels and learns to choose the better performing version. In contrast to our system, the various versions are not organized in a hierarchy and our system does not have to learn which version to choose, but automatically chooses the most specific version for the many-core device.

Bueno et al. report slightly higher performance on 8 GTX480 nodes for Matrix Multiplication (just above 3 TFLOPS compared 2.8 TFLOPS for the optimized kernel for Cashmere). However, they use a CUBLAS kernel that has higher initial performance than our optimized kernel.

StarPU [21] provides an execution model based on tasks. Programmers implement tasks in the native many-core programming system, for example CUDA or a BLAS routine and they annotate them with the tasks on which they depend. The run-time system handles data-dependencies, scheduling of tasks, and load balancing. Experiments in [21] have been performed on much older GPUs making a comparison difficult.

SkePU [22] is built on top of StarPU and provides a skeleton framework that allows programmers to express a program in terms of frequently used patterns that are encoded in generic constructs. The program remains sequential while the implementation of the patterns is parallel. Since the paper only reports relative speedups, comparing performance is difficult. However, SkePU does not appear to scale as well as Cashmere: for the best configuration of N-body, SkePU obtains a speedup of about 2.6 on 3 GPU nodes.

PaRSEC [23], [24] is a framework with a task-based run-time to overcome the problem that MPI does not scale well for more dynamic applications. Programmers have to manually annotate data-flow dependencies between tasks with their granularity and PaRSEC does not have an integrated solution for programming kernels as Cashmere has. However, PaRSEC provides a more general programming model than divide-and-conquer. PaRSEC only has a CUDA back-end and was not evaluated with heterogeneous many-core devices.

VII. Conclusion

Heterogeneity is becoming the norm in clusters and data-centers as a result of performance and power-consumption benefits of new generations of many-core hardware. This poses several challenges for the already demanding task of programming many-core clusters. Our solution, Cashmere, seamlessly integrates many levels of parallelism. It includes a framework to write and optimize kernels for different many-core devices in which common optimizations can be shared thanks to the support for multiple abstraction levels. Our approach delivers high performance and automatic load balancing even when the many-core devices differ widely. Cashmere achieves high efficiency (>90% in three out of four applications) in heterogeneous executions.

References

[1] B. van Werkhoven, J. Maassen, H. E. Bal, and F. J. Seinstra, "Optimizing convolution operations on GPUs using adaptive tiling," *Fut. Gen. Comp. Systems*, vol. 30, pp. 14 – 26, 2014.

[2] DAS-4: Distributed ASCI Supercomputer 4. [Online]. Available: http://www.cs.vu.nl/das4

[3] (2015, Jan) TOP500 Supercomputer Sites. [Online]. Available: http://www.top500.org

[4] J. Stone, D. Gohara, and G. Shi, "OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems," *Comp. in Science & Eng.*, vol. 12, no. 3, pp. 66–73, 2010.

[5] J. Nickolls, I. Buck, M. Garland, and K. Skadron, "Scalable Parallel Programming with CUDA," *Queue*, vol. 6, no. 2, pp. 40–53, 2008.

[6] R. V. van Nieuwpoort, G. Wrzesińska, C. J. H. Jacobs, and H. E. Bal, "Satin: A High-Level and Efficient Grid Programming Model," *ACM Trans. Program. Lang. Syst.*, vol. 32, no. 3, pp. 1–39, 2010.

[7] P. Hijma, R. V. van Nieuwpoort, C. J. Jacobs, and H. E. Bal, "Stepwise-refinement for performance: a methodology for many-core programming," *Concurrency and Computation: Practice and Experience*, pp. n/a–n/a, 2015. [Online]. Available: http://dx.doi.org/10.1002/cpe.3416

[8] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou, "Cilk: An Efficient Multithreaded Runtime System," *SIGPLAN Not.*, vol. 30, no. 8, pp. 207–216, 1995.

[9] K. Beason. (2008) smallpt: Global illumination in 99 lines of c++. [Online]. Available: http://www.kevinbeason.com/smallpt

[10] D. Bucciarelli. (2009) Smallptgpu. [Online]. Available: http://davibu.interfree.it/opencl/smallptgpu/smallptGPU.html

[11] P. Xiang, Y. Yang, and H. Zhou, "Warp-Level Divergence in GPUs: Characterization, Impact, and Mitigation," in *Int. Symp. on High Perf. Comp. Arch. (HPCA)*, 2014, pp. 284–295.

[12] J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," *Commun. ACM*, vol. 51, no. 1, pp. 107–113, Jan. 2008.

[13] J. A. Stuart and J. D. Owens, "Multi-GPU MapReduce on GPU Clusters," in *Int. Par. and Dist. Proc. Sym. (IPDPS)*. Los Alamitos, CA, USA: IEEE Comp. Society, 2011, pp. 1068–1079.

[14] M. Grossman, M. Breternitz, and V. Sarkar, "HadoopCL: MapReduce on Distributed Heterogeneous Platforms Through Seamless Integration of Hadoop and OpenCL," in *IPDPSW '13*. Washington, DC, USA: IEEE Computer Society, 2013, pp. 1918–1927.

[15] T. White, *Hadoop: The Definitive Guide*, 1st ed. O'Reilly Media, Inc., 2009.

[16] G. Frost. (2011) APARAPI: API for data parallel Java. https://code.google.com/p/aparapi.

[17] I. El-Helw, R. Hofman, and H. E. Bal, "Scaling MapReduce Vertically and Horizontally," in *SC '14: Proc. of the 2014 ACM/IEEE conf. on Supercomputing*. ACM, 2014.

[18] J. Bueno, J. Planas, A. Duran, R. M. Badia, X. Martorell, E. Ayguade, and J. Labarta, "Productive Programming of GPU Clusters with OmpSs," in *Int. Par. and Dist. Proc. Sym. (IPDPS)*. Los Alamitos, CA, USA: IEEE Comp. Society, 2012, pp. 557–568.

[19] J. Perez, R. Badia, and J. Labarta, "A Dependency-Aware Task-Based Programming Environment for Multi-Core Architectures," in *IEEE Int. Conf. on Cluster Computing*, Sep. 2008, pp. 142 –151.

[20] J. Planas, R. M. Badia, E. Ayguadé, and J. Labarta, "Self-Adaptive OmpSs Tasks in Heterogeneous Environments," in *Int. Par and Dist. Proc. Sym. (IPDPS)*. Washington, DC, USA: IEEE Computer Society, 2013, pp. 138–149.

[21] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier, "StarPU: a unified platform for task scheduling on heterogeneous multicore architectures," *Concurrency and Computation: Practice and Experience*, vol. 23, no. 2, pp. 187–198, 2011.

[22] M. Majeed, U. Dastgeer, and C. Kessler, "Cluster-SkePU: A Multi-Backend Skeleton Programming Library for GPU Clusters," in *Proc. of the Int. Conf. on Par. and Dist. Proc. Techn. and Appl. (PDPTA)*, Las Vegas, USA, July 2013.

[23] G. Bosilca, A. Bouteiller, A. Danalis, M. Faverge, T. Herault, and J. Dongarra, "PaRSEC: Exploiting Heterogeneity to Enhance Scalability," *Computing in Science Engineering*, vol. 15, no. 6, pp. 36–45, Nov 2013.

[24] G. Bosilca, A. Bouteiller, A. Danalis, T. Herault, P. Luszczek, and J. J. Dongarra, "Dense Linear Algebra on Distributed Heterogeneous Hardware with a Symbolic DAG Approach," in *Scalable Computing and Communications: Theory and Practice*, S. U. Khan, L. Wang, and A. Y. Zomaya, Eds. John Wiley & Sons, Jan 2013, pp. 699–733.