MANY-TASK COMPUTING ON MANY-CORE ARCHITECTURES

PEDRO VALERO-LARA,* POORNIMA NOOKALA,† FERNANDO L. PELAYO,‡ JOHAN JANSSON § SERAPHEIM DIMITROPOULOS,¶ AND IOAN RAICU‖

Abstract. Many-Task Computing (MTC) is a common scenario for multiple parallel systems, such as cluster, grids, cloud and supercomputers, but it is not so popular in shared memory parallel processors. In this sense and given the spectacular growth in performance and in number of cores integrated in many-core architectures, the study of MTC on such architectures is becoming more and more relevant. In this paper, authors present what are those programming mechanisms to take advantages of such massively parallel features for the particular target of MTC. Also, the hardware features of the two dominant many-core platforms (NVIDIA's GPUs and Intel Xeon Phi) are also analyzed for our specific framework. Given the important differences in terms of hardware and software in our two many-core platforms, we have considered different strategies based on CUDA (for GPUs) and OpenMP (for Intel Xeon Phi). We carried out several test cases based on an appropriate and widely studied problem for benchmarking as matrix multiplication. Essentially, this study consisted of comparing the time consumed for computing in parallel several tasks one by one (the whole computational resources are used just to compute one task at a time) with the time consumed for computing in parallel the same set of tasks simultaneously (the whole computational resources are used for computing the set of tasks at very same time). Finally, we compared both software-hardware scenarios to identify the most relevant computer features in each of our many-core architectures.

Key words: Parallel Computing, Multi-Task Computing, Many-Core, GPU, Intel Xeon Phi, CUDA, OpenMP

AMS subject classifications. 15A15, 15A09, 15A23

1. Introduction. Many-Task Computing (MTC), the execution of multiple tasks on one particular parallel platform at very same time, is historically dominated by some parallel platforms such as clusters, grids, and supercomputers. However, the advances in hardware, in particular in many-core architectures, for MTC applications is a relevant topic. Again, the main problem is at the software side. Programmers need to address the challenge of analyzing and studying the different hardware features to efficiently map the MTC applications in order to achieve the best performance on such architectures.

The main contribution of the present work is twofold. While, on one hand the first motivation consists of presenting those approaches and mechanisms to efficiently exploit Multi-Task Computing applications on current many-core architectures. Secondly, but not least important, authors provide a study to clarify what are the most amenable features of the two dominant many-core systems today, these are NVIDIA GPUs and Intel Xeon Phi, for the specific target of MTC.

In the last years, the use of scheduler based on many-core or heterogeneous architectures for general or for specific applications has been widely studied [26, 48]. S. Yamagiwa et al. [48] propose a GPGPU streaming based on distributed computing environment; S. Nakagawa et al. [26] provide a new middleware capable of out-of-order execution of works and data transfers using stream processing. Other works [12, 46] follow a similar strategy based on streaming to minimize data transfers overhead. S. Kato et al. [21] introduce *TimeGraph*, a GPU scheduler composed by two different GPU scheduling policies which allow to interrupt the low priority tasks execution in order to execute higher priority tasks within a real-time multi-tasking environments for video applications. Similar to the previously mentioned works and considering that the GPUs in a cluster are not usually fully utilized, Duato et al. [9] present their *rCUDA*, a middleware that enables CUDA remoting over a commodity network by allowing to use CUDA-compatible GPUs installed in a remote computer, as, they were installed in the computer where the application is being executed. Also, V. J. Jiménez et al. [19] present a sort of predictive runtime scheduling which supports several scheduling algorithms in order to choose the appropriate platform (Multicore, GPU, etc.) in which the algorithm would be better executed, resulting in

---

*Univeristy of Manchester, UK, and Basque Center for Applied Mathematics (BCAM), Bilbao, Spain (pvalero@bcamath.org).

†Illinois Institute of Technology (IIT), Chicago, USA (pnookala@hawk.iit.edu.)

‡University of Castilla-La Mancha (UCLM), Albacete, Spain (fernandol.pelayo@uclm.es.)

§Basque Center for Applied Mathematics (BCAM), Bilbao, Spain, and KTH Royal Institute of Technology, Stockholm, Sweden (jjansson@bcamath.org).

¶Illinois Institute of Technology (IIT), Chicago, USA (sdimitro@hawk.iit.edu.)

‖Illinois Institute of Technology (IIT), Chicago, USA (iraicu@cs.iit.edu.)

almost fully usage of CPU/GPU-like systems, with a peak time reduction of 40% with respect to only using the GPU. Basically most the aforementioned works take advantage of overlapping memory transfers among CPU and GPU memories with single kernel executions.

With the aim of exploiting MTC on many-core, other authors [23, 25] have studied the efficiency of this new feature. *Batched task*, maybe the first MTC approach on GPUs, allows us to run several independent kernels over the same GPU simultaneously. It was presented by M. Guevara et al. [14] and P. Valero-Lara et al. [41]. Posteriorly, C. Gregg et al. [13] and K. Zhang et al. [49] included a scheduler which can select the best matching among tasks before running. Additionally, P. Valero-Lara et al. [42] applied this strategy to different GPU architectures to obtain the most convenient architectural features for running concurrent kernels. After that, in [40], it is proposed a new heterogeneous (CPU-GPU) scheduler in which groups of independent blocks of tasks were efficiently managed to fully use CPU-GPU and reduce the overhead of memory transfers. More recently, S. Krieder et al. [24] presented *GeMTC*, a CUDA based framework which allows MTC workloads to run efficiently on NVIDIA's GPUs. Posteriorly P. Nookala et al. [27] adapted this framework (*GeMTC*) to efficiently use the particular features of Intel Xeon Phi and evaluate MTC applications on Intel accelerators. In fact, we can now find some applications which takes advantage of MTC on hardware accelerators. One of these applications was presented by P. Valero-Lara et al. [43, 44], in which multiples tridiagonal problems are efficiently executed on the same NVIDIA's GPU simultaneously. Other examples consist of computing several relatively small Linear Algebra problems [16, 15, 17], multiple range queries in metric spaces [2, 1] or multiple string matching [22].

This work is structured as follows: Section 2 introduces the main features of many-core architectures considered (NVIDIA GPUs and Intel Xeon Phi), then in Section 3, authors briefly outline the different mechanisms for MTC on both many-core architectures. After that, both platforms and the MTC mechanisms are deeply analyzed and studied. Finally, Section 5 concludes summarizing the most relevant results.

2. Many-Core Architectures. Today, the increase in performance for single-threaded processor has come to an end due to the limitation of the current Very Large Scale Integration (VLSI) technology. In response, most hardware companies are designing and developing new parallel architectures [11]. Programs will only increase in performance if they use and exploit the new parallel characteristics of new architectures. On the other hand, multicore designs are also encountering scaling problems, notably the "Dark Silicon" phenomenon [10]. Power and cooling concerns suggest the number of dynamically active transistors on a single die may be greatly constrained in the near future. In other words, even if the number of transistors per chip continues to follow Moore's law, we will not be able to use all of them simultaneously. This problem may lead to scenarios in which only a small percentage of the chip's transistors can be "on" at a time [34]. Given the limitations of current CMOS technology and the excessive power consumption reached by current platforms, it is necessary a renewal of hardware design.

In this context, many-core architectures may be an answer to these challenges. These new massively parallel platforms offer a high ratio performance/cost and an efficient power consumption design [39, 38, 37]. They are also widely used on high performance computing, including systems ranging from cluster of personal computers, to large scale supercomputers.

Most processors for high-performance computing (HPC) are still multi-core. However, as we can see in Top 500 list [36], many of the most powerful supercomputers today are based on platforms that combine multicore processors with data parallel accelerators. The fastest system, which is currenlty the Tianhe-2 supercomputer from China, uses Intel's Xeon Phi coprocessors and its runnerup, which is the Titan supercomputer from the Oak Ridge National Laboratory, uses NVIDIA GPUs.

2.1. Graphic Processing Units (GPUs). GPUs are traditionally used for interactive applications, and are designed to achieve high rasterization performance however, their characteristics have allowed the opportunity to other more general applications to be accelerated in GPU-based platforms. This trend is now called General Purpose Computing on GPU (GPGPU) [31], or what is the same, the usage of GPUs for applications for which they were not originally designed. These general applications must have parallel characteristics and an intense computational load to obtain a good performance.

The main feature of these devices is a large number of processing elements integrated into a single chip at the expense of a significant reduction in cache memory. These processing elements are arranged on memory
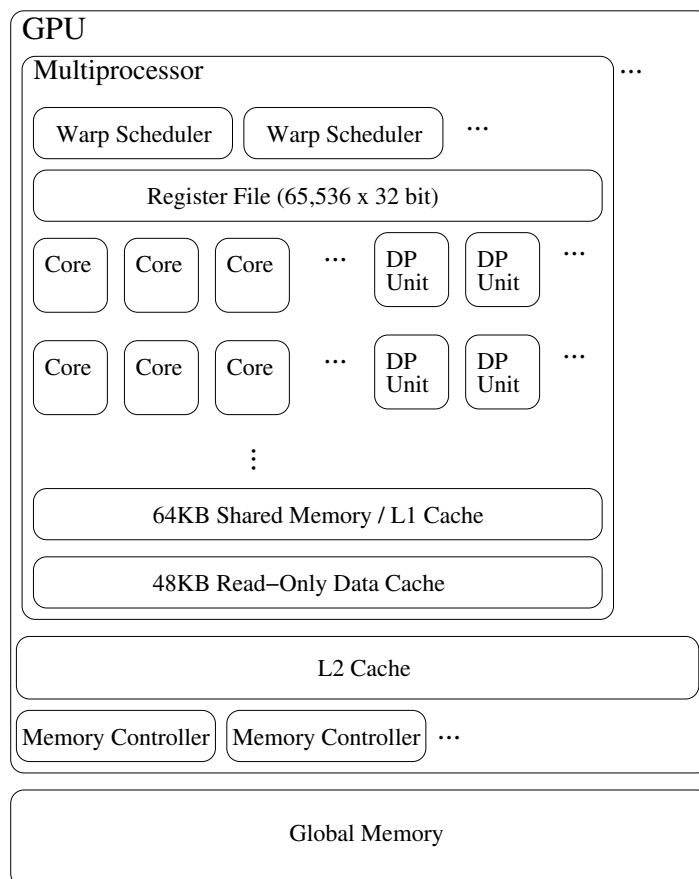
Fig. 2.1: NVIDIA GPU (Kepler) architecture [45].

cards that have a local high-speed external DRAM and are connected to the computer through a high-speed I/O interface (PCI Express).

Figure 2.1 shows an abstract block diagram of NVIDIA's (Kepler) GPU [45]. The GPU is organized into several multiprocessors, which in turn are composed of various simple processors (cores) that operates in SIMD fashion. The multiprocessors have fine grain multithreading capabilities, which means that they support hundreds of threads in-fly. Every multiprocessor switches to a different set of threads every clock cycle, which helps to maximize computational resources and hide the long latency memory accesses to a share GPU main memory.

The GPU main memory, usually called "global memory", is banked, which allows the hardware to coalesce several simultaneous memory accesses to adjacent positions into a single memory transaction. In addition, each multiprocessor contains a large set of registers and an on-chip SRAM scratchpad memory, i.e., a software controlled cache, to speed up data access. In more recent GPUs (starting from NVIDIA's Fermi architecture) the SRAM can be configured either as scratchpad or as cache memory and the user decide, with certain restrictions, the size of both memories. These newer GPUs also incorporate a L2 cache common to all multiprocessors. The access to the global memory can also be performed through special read-only two level hierarchy of so called texture caches, that are optimized to capture 2D access patterns [47].

2.2. Intel Xeon Phi. GPUs have a very restrictive programming model, but provide at least an order of magnitude better throughput for applications painstakingly coded to that model. To program GPUs, typically there is a need to learn another programming language such as CUDA (NVIDIA) or OpenCL (AMD). As
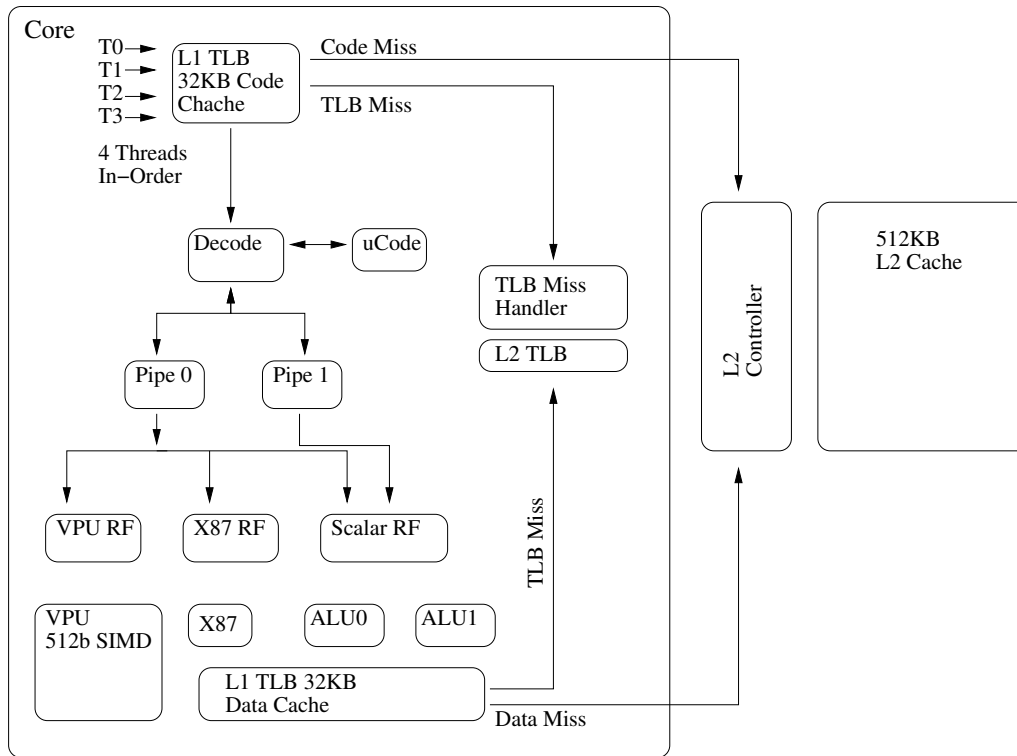
Fig. 2.2: Architecture of a single Intel Xeon Phi Core [18].

a result, existing vendors must spend extra time and effort to modify or rewrite parts of their codebase to take advantage of the new capabilities provided by General Purpose GPUs (GPGPUs). Besides that, barely rewriting an application just to offload computations to a GPU rarely works well. Because of the architecture of most GPUs out there, applications must be tailored from the ground up to follow the rules of the restrictive programming model of GPUs, otherwise they may suffer from severe performance penalties. Because of that, interested vendors cannot afford to go through the effort involved. Finally, while GPUs are great for massively parallel applications with thread- switching that comes almost at no cost, their performance can take a large hit when executing programs with complex logic (like complicated branching and looping for example). Therefore they may be unsuitable for certain applications of MTC. The Intel Xeon Phi is a new family of processors based on the Intel MIC Architecture [18] that incorporates earlier work on the Larrabee architecture [33]. It follows an alternative programming model that, although may not provide the same level of parallelism, provides more flexibility and therefore can be more suitable for certain application of MTC that GPUs are not suited for. The reason is that the Xeon Phi has x86 cores that are more capable (can handle complex branching and looping) than most GPU cores. Another advantage of having x86 cores is that programming the coprocessor minimizes the amount of work that needs to be done in order to integrate a Xeon Phi to an existing system. That is because the Phi does not require being programmed in any specific framework and it can natively run applications written in C with Pthreads or OpenMP. All of the above facts were enough to motivate us to work on this project. We have used the 22nm Knights Corner chip graphically described in Figures 2.2 and 2.3, which was the first commercial product from this family.

The Corner is a PCIe vector co-processor with integrates up to 61 in-order dual issue x86 cores, which trace some history to the original Pentium core, like the Larrabee predecessor. Among other enhancements, the Corner's cores are augmented with 64-bit support, 4 hardware threads per core (resulting in more than 200 hardware threads available on a single device) and 512-bit SIMD instructions [18]. Each core has a 512KB
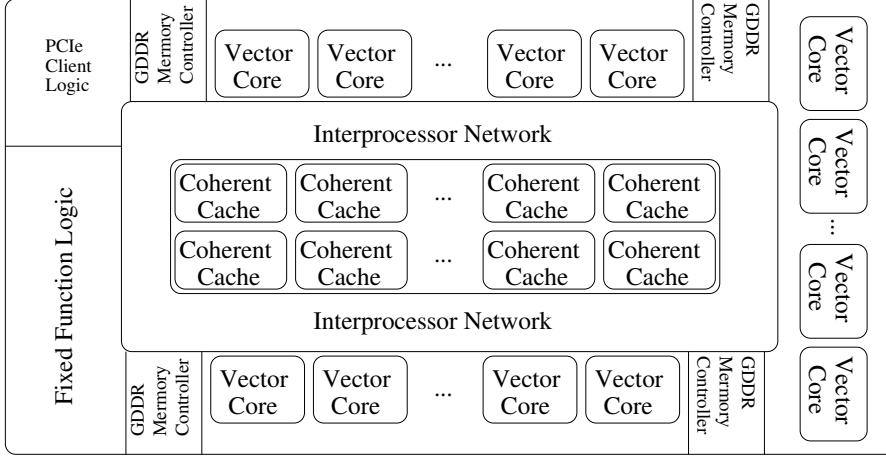
Fig. 2.3: Micro-architecture of the Entire MIC coprocessor [18].

L2 cache locally but has also access to all other L2 caches in the system through a high-speed bidirectional ring [18]. Unlike previous GPUs, the L2 cache is kept fully coherent by a global-distributed tag directory.

The performance achieved by Knight Corner chips is usually outperformed by NVIDIA's counterparts [29]. However, last year Intel announced the Knight Landing processor [35] that should significantly improve MIC performance.

3. Multi-Task Computing (MTC). Although, the dominant choice to compute MTC problems continues being distributed memory architectures, the impressive growth in performance of current parallel shared memory architectures makes possible to compute a considerable high number of independent tasks over this kind of computational platforms.

In this regard, this section introduces some of the most extended and known approaches for computing MTC over many-core architectures (NVIDIA GPUs and Intel Xeon Phi). Essentially, each of these approaches share the same major steps (Figure 3.1). These consist of performing memory transfers (communication) sequentially, then the set of independent tasks are executed on many-core platform.

3.1. MTC on GPU. This subsection introduces 3 different approaches for MTC running on NVIDIA's GPUs. To clarify, we include some pseudocodes which can help us to understand the differences among them and the particular features (advantages and disadvantages) of every of them.

Batched Task, several authors [13, 14, 41, 42] proposed this strategy to fully utilize the GPU by running multiple tasks (kernels) simultaneously on the same GPU. Basically, all of them include a single pass compiler which is able to create the batched task source code by renaming the variables, by adding the if-else control flow, and by adding indexing in the independent task that is executed by blocks that are offset from *blockId*. The set of tasks are mapped either on one or on a set of blocks of threads. In this case, the number of threads launched must be equal than the sum of all threads required by all tasks. Also, all parameters must be included in the same call. We would like to point out that this strategy can be carried out on all CUDA GPUs architectures. Algorithm 1 illustrates a simple scheme of this strategy.

NVIDIA proposed a new approach [5, 40] called **Concurrent Kernels** which allows to execute a set of independent tasks (kernels) on the same GPU by means of streams. It only can be used over *FERMI* architecture forward. Using different streams for CUDA kernels makes the concurrent execution being possible. Therefore n kernels on n streams could theoretically run concurrently if they are fitted into the hardware. This approach allows to execute up to 16 different kernels at the very same time. A scheme of this approach is shown in Algorithm 2, where two kernels are used in such way.

Recently, NVIDIA has introduced a new feature compatible for the *KEPLER* CUDA architecture (**Dy-**
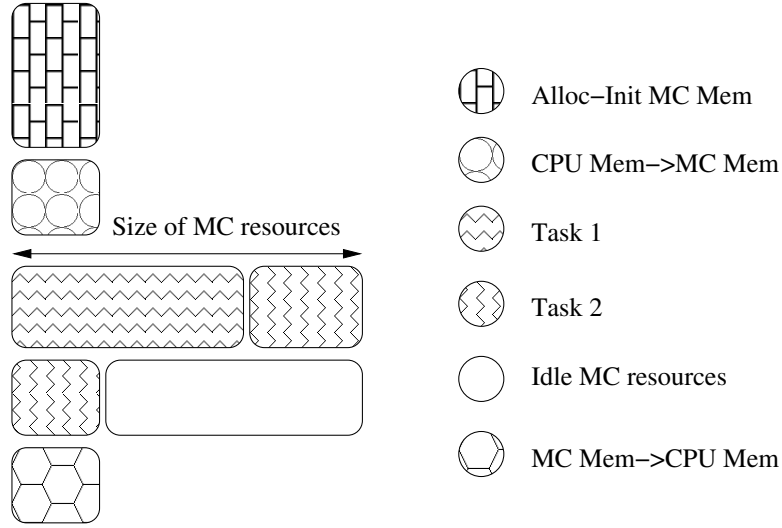
Fig. 3.1: Basic scheme for MTC on Many-Core (MC) architectures. Allocation and Initialization of MC memory (Alloc-Init MC Mem). Data transfer (input) from CPU memory to MC memory (CPU Mem $\rightarrow$ MC mem). MC executions (Task 1 and Task 2). Idle cores in MC (Idle MC resources). Data transfer (output) from MC memory to CPU mem (MC Mem $\rightarrow$ CPU Mem).

Algorithm 1 Batched Task.

    `BatchedTaskCPU`
1: CPUMemAllocate(A_{CPU},B_{CPU},C_{CPU},D_{CPU})
2: GPUMemAllocate(A_{GPU},B_{GPU},C_{GPU},D_{GPU})
3: CPU->GPUMemTransfer(A_{CPU},A_{GPU})
4: CPU->GPUMemTransfer(C_{CPU},C_{GPU})
5: CPU->GPUMemTransfer(D_{CPU},D_{GPU})
6: BatchedTask<THREADS1+THREADS2> (A_{GPU},B_{GPU},C_{GPU},D_{GPU})
7: GPU->CPUMemTransfer(B_{GPU},B_{CPU})
8: GPU->CPUMemTransfer(D_{GPU},D_{CPU})
    `BatchedTaskGPU(A,B,C,D)`
9: i = index of thread
10: j = index of block
11: **if** j = 0 **then**   ▷ kernel1
12:     B[i] = A[i] + 100
13: **else if** j = 1 **then**   ▷ kernel2
14:     D[i] = C[i] × D[i]
15: **end if**

---

namic Parallelism [6]) which allows to manage multiple tasks inside GPU. It is supported via an extension of the CUDA programming model that enables a CUDA kernel to create and to synchronize new nested work. CUDA kernel can consume the output from the other kernels (childs) without CPU involvement. It requires at least two-level task running (parent-child).

3.2. MTC on Intel Xeon Phi. Due to the foundations of Intel architecture, the coprocessor can be programmed in several different ways [32]. Here we introduce two different approaches, one using *OpenMP* and one using *SCIF* (Intel's Symmetric Communication Interface). *OpenMP* implementation uses offloading approach for offloading computations from host to the accelerator. The *SCIF* implementation runs natively

---

Algorithm 2 Concurrent kernels.

```
ConcurrentKernelsCPU
```
 1: CUDAStream Stream[2]
 2: CPUMemAllocate(A_{CPU},B_{CPU},C_{CPU},D_{CPU})
 3: CPU->GPUMemTransfer(A_{CPU},A_{GPU})
 4: CPU->GPUMemTransfer(C_{CPU},C_{GPU})
 5: CPU->GPUMemTransfer(D_{CPU},D_{GPU})
 6: **for** $i = 1 \rightarrow 2$ **do**
 7:     StreamCreate(Stream[i])
 8: **end for**
 9: Kernel1<THREADS1>(A_{GPU},B_{GPU},Stream[1])
10: Kernel2<THREADS2>(C_{GPU},D_{GPU},Stream[2])
11: **for** $i = 1 \rightarrow 2$ **do**
12:     StreamDestroy(Stream[i])
13: **end for**
14: GPU->CPUMemTransfer(B_{GPU},B_{CPU})
15: GPU->CPUMemTransfer(D_{GPU},D_{CPU})

Algorithm 3 Dynamic parallelism.

```
DynamicParallelismCPU
```
 1: CPUMemAllocate(A_{CPU},B_{CPU},C_{CPU},D_{CPU})
 2: GPUMemAllocate(A_{GPU},B_{GPU},C_{GPU},D_{GPU})
 3: CPU->GPUMemTransfer(A_{CPU},A_{GPU})
 4: CPU->GPUMemTransfer(C_{CPU},C_{GPU})
 5: CPU->GPUMemTransfer(D_{CPU},D_{GPU})
 6: DynamicParallelismKernel<1> (A_{GPU},B_{GPU},C_{GPU},D_{GPU})
 7: GPU->CPUMemTransfer(B_{GPU},B_{CPU})
 8: GPU->CPUMemTransfer(D_{GPU},D_{CPU})
```
DynamicParallelismKernel(A,B,C,D)
```
 9: Kernel1<THREADS1>(A_{GPU},B_{GPU})
10: Kernel2<THREADS2>(C_{GPU},D_{GPU})
11: SyncronizeThreads

---

on the accelerator and accepts jobs from Clients running on the host. There are several advantages and disadvantages between the two methods. The major advantage of native execution coupled with *SCIF* over offloading is that the developer gets more control overall in the configuration and the architecture of their design in order to maximize performance. Computation does not necessarily have to be transferred back to to the CPU. In addition, different MIC cards can communicate directly with each other basically making certain designs more efficient.

Finally, frameworks that use offloading mode (*OpenMP*), do not necessarily take advantage of the DMA-features of the hardware they run on while on *SCIF* you are guaranteed that if you are using Remote Memory Access (RMA). That is not to say that *OpenMP* does not come with any advantages over *SCIF*. Quite the opposite, the advantages of offloading are pretty significant for the framework that was implemented for this project. The low-level C code needed for the *SCIF* implementation is relatively a lot more complex when compared with *pragma* directives provided by *OpenMP*. In addition, using *SCIF* implies that the framework must have at least one of its parts running natively on the Phi as the endpoint. In order to do that the developer needs to set up an application to run natively on the Phi and involves a lot of configuration. Using *OpenMP* with the offloading capabilities provided by the MIC, all this configuration is taken care of.

OpenMP version (Figure 3.2) uses asynchronous offloading capabilities. We have employed a Producer-Consumer architecture which communicates using shared memory for IPC (InterProcess Communication). The
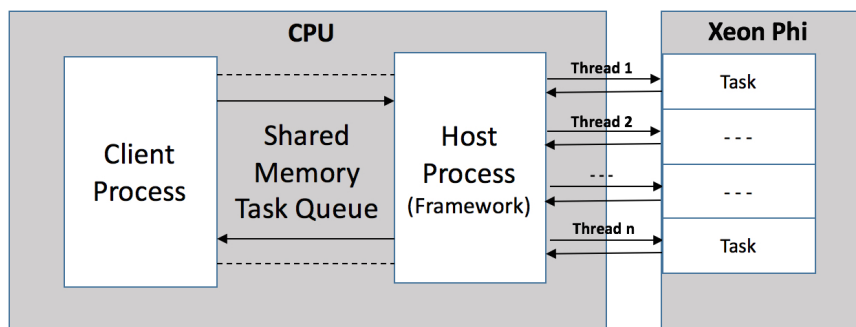
Fig. 3.2: OpenMP-Phi diagram.

Consumer side hosts the framework which runs as a single thread and launches as many master processes on the Xeon Phi as specified by the user [28]. The master processes use the shared memory space as a queue structure, continuously accepting new tasks from producer processes. Likewise, the producer acts as a client process which submits tasks to the queue via this shared memory pool. For testing this framework, we have implemented two different types of applications: Sleep and Matrix Multiplication. Both were developed and tested using the OpenMP approach of offloading tasks on to Xeon Phi. The master processes in our framework read the tasks from shared memory location and based on the type of task, offload the computation part to Xeon Phi. Asynchronous offloading is used to allow the framework to continue accepting tasks while other tasks are running. The Phi sends a signal back to the master processes after job execution has completed. At this point the output is sent back to the Client.

SCIF implementation employs a Client Server architecture [4] where clients send their tasks to the Phi from the host and the server, which runs natively on the Phi, accepts the jobs. After submitting the job, the clients can request the result and the server will deliver it to them when the task has finished processing and is placed on the results queue of the framework. The whole procedure is non-blocking for the server who can handle multiple requests and submissions at the same time. That functionality is implemented with *epoll()* [30] for handling connections that are later passed to threads [20, 24] that push or dequeue tasks from the queues. The *SCIF* socket-like API is used for communications between the server and the clients. It comes as a shared library named **libmtcq* [8]. This library includes all the functionality that handles incoming and outgoing queues of tasks, pushing jobs and distributing tasks to workers. It is also completely parametrizable in terms of queue sizes, worker threads, and application threads. Since the Xeon Phi does not have the hierarchical architecture of SMXs and Warps nor the concept of application kernels that you generally see in GPGPUs, everything is implemented with standard Pthreads. There is a parametrizable number of master threads that dequeues tasks from the incoming queue. If the task is a parallel application, which is the case most of the time, then the master thread will assign the task to the specified number of worker threads. Else if it is sequential only one thread will be assigned and the master thread will go back to dequeue more jobs. Each queue is implemented as a finite buffer from the Producer-Consumer model which means that it uses a single mutex and two semaphores to ensure that no deadlocks or data-races arise.

4. Performance Evaluation. This section presents a performance study to test and obtain what are the most relevant programming and hardware features for MTC. Each of the programming approaches and many-core architectures are analysed in deep.

4.1. NVIDIA GPUs. Taking into account that most many-core accelerators, especially GPUs, reach their optimum performance over problems having a high level of data parallelism together with a fine granularity, we have planned a set of tests over one suitable problem (in the previous sense) as matrices multiplication. The implementation used is an optimization of SGEMM method on GPU presented in [3] which incorporates
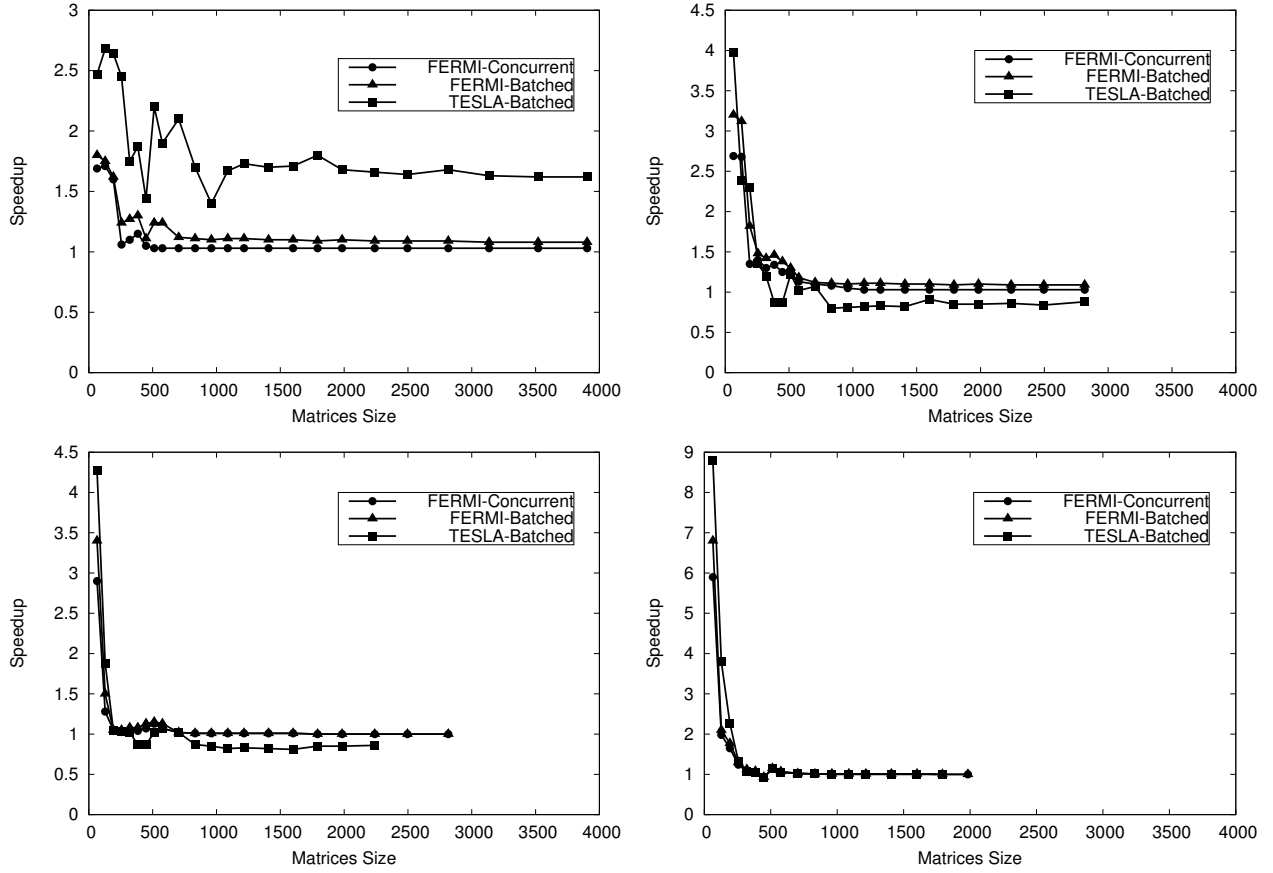
Fig. 4.1: Performance (speedup) achieved by the *Batched* (TESLA and FERMI architectures) and *Concurrent* (FERMI architecture) approaches.

several optimizations for NVIDA's GPUs, such as coalescing memory accesses and shared memory exploitation. Besides, we have considered the best choice concern with the size of threads blocks.

Four different test cases have been carried out which consist of computing 2, 4, 8 and 16 tasks (maximum number of tasks for FERMI architecture) in parallel on the same GPU. The size of matrices is increased in order to show the impact on performance by increasing memory requirements. Results are shown in terms of speedup (Figures 4.1 and 4.2), which is the ratio between the execution time when running (one by one) several tasks (matrix multiplication), and, the execution time when computing all multiplications (MTC approach) on the same GPU in parallel (Figure 3.1) according to each approach, *Dynamic, Concurrent* and *Batched*. Due to the memory capacity of TESLA and FERMI, some tests carried out in the first graph (Figure 3.1-Top) could not be included in the rest.

We have considered three different NVIDIA GPU architectures which we have re-called as TESLA (GT 200), FERMI (GF 100) and KEPLER (GK 110). Although, all of them share the major components, we can find important differences (Table 4.1). As we see later, these differences have important consequences in performance. Each of the CUDA-compatible MTC approaches (subsection 3.1) have been tested on our three GPU architectures, when it is possible. The study carried out in this subsection is an extension of the work previously presented in [41]. We include additional results using new MTC approaches on new GPU architectures. We evaluate the *Batched* and *Concurrent* approaches on our FERMI, as the *Dynamic* one is not compatible with this architecture. Otherwise, we can analyze all approaches on KEPLER GPU, as it is one of
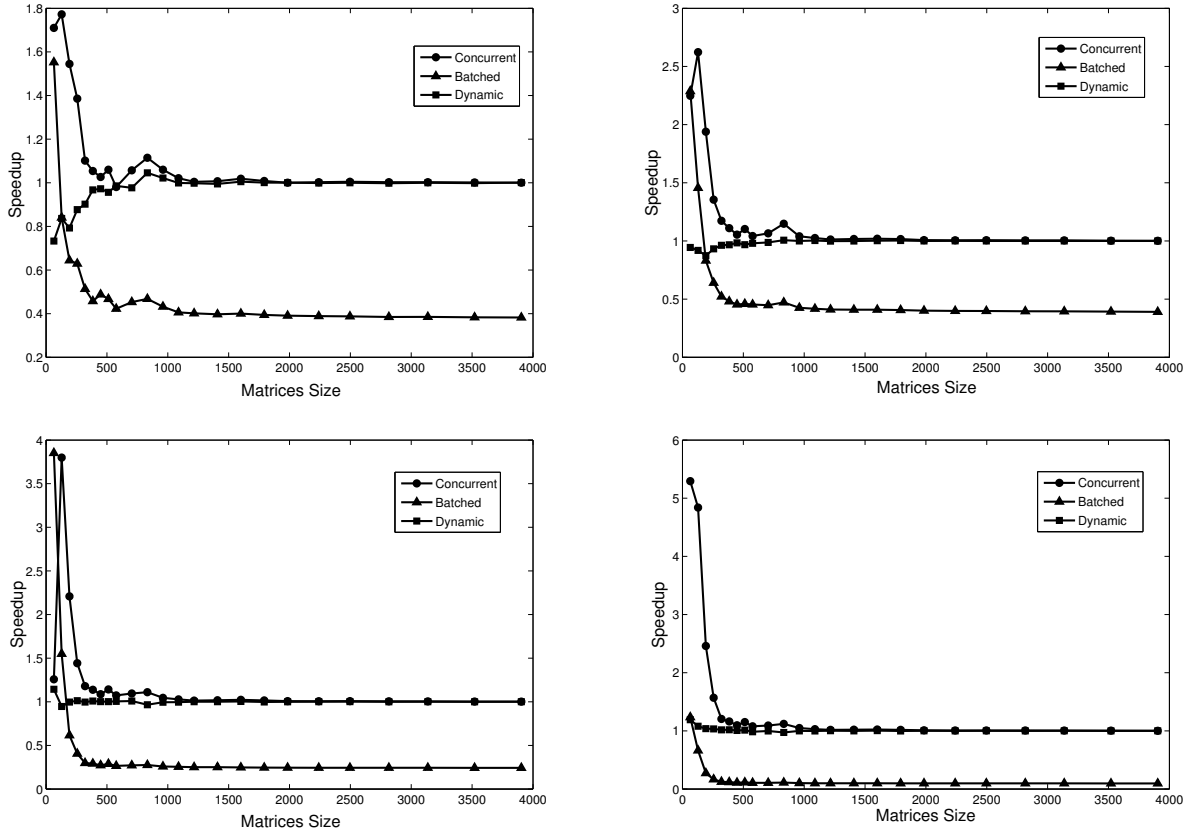
Fig. 4.2: Performance (speedup) achieved by each of the approaches (*Batched*, *Concurrent* and *Dynamic*) over KEPLER GPU architecture.

the newest NVIDIA GPUs. However, our TESLA GPU is only compatible with the *Batched* approach, as it is oldest architecture included in our study.

From the results obtained (Figures 4.1 and 4.2), we highlight several conclusions comparing each of the MTC approaches over each of the architectures. On FERMI both approaches, *Batched* and *Concurrent*, achieve a similar result, being faster the *Concurrent* one. Although the *Dynamic* scheduler is an easy to implement approach from programmer point of view, it does not show any benefit on KEPLER. In contrast, a better scalability is reached by using the other two approaches. The *Batched* approach presents a good scaling for small matrix sizes, however, it turns to be inefficient for bigger sizes. Otherwise, the best scaling is obtained by the *Concurrent* approach, even for medium matrix sizes.

Both approaches, *Batched* and *Concurrent*, share a similar trend on each of the architectures. First, we focus on the impact of matrix size in performance. In this regard, the best performance is reached in the first tests (small matrices). Obviously, the performance achieved on the first tests is degraded by increasing matrix what implies to increase also the number of threads per CUDA block. As consequence, the GPU resources (multiprocessors) are saturated so that a higher degree of parallelism can not be efficiently exploited. Second, we focus on analysing the trend in performance by increasing the number of tasks. Unlike the results achieved by increasing the matrix size, in which we appreciate an important fall in performance, we see the opposite scenario, that is the performance achieved is higher by increasing the number of tasks, at least for small matrix sizes.

Although all architectures are similar and share the major components, the most relevant variance among

Table 4.1: GTX 285, GTX 480 and K 20c hardware.

	GTX 285	GTX 480	K 20c
Code Name	GT 200(TESLA)	GF 100(FERMI)	GK 110 (KEPLER)
# Multiprocess. (MP)	30	15	13
# Cores/MP	8	32	192
# Cores	240	480	2496
Core Clock	648 Mhz	1401 Mhz	706 Mhz
Mem. Clock	1242 Mhz	1848 Mhz	2600 Mhz
Mem. Capacity	1 GB	1.5 GB	5 GB
On-chip Mem.			
SM (per MP)	-	16/48 KB	16/48 KB
L1 (per MP)	-	48/16 KB	48/16 KB
L2 (unified)	-	768 KB	768 KB
Mem. Bus	512 bits	384 bits	320 bits
Bandwidth	159 GB/s	177.4 GB/s	208 GB/s
Gigaflops (SP)	708	1344.96	4577

Table 4.2: Intel Xeon Phi hardware.

	Intel Xeon Phi
Code Name	5110P (PHI)
# Cores	60
Core Clock	1053 Mhz
Mem. Capacity	8 GB
On-chip Mem.	
L1 (per core)	32 KB
L2 (per core)	256 KB
L2 (coherent)	30 MB
Bandwidth	320 GB/s
Gigaflops (DP)	2022

them is found in the number of multiprocessors (Table 4.1). While KEPLER is composed by 13 multiprocessors, TESLA is composed by 30. Given these results, we can assume that this factor has a great influence in performance. Although the number of cores in KEPLER is more than $10\times$ than in TESLA, it is remarkable, that the speedup reached by TESLA is up to almost $2\times$ bigger than the speedup obtained by KEPLER. In this regard, the number of cores is not as relevant as the number of multiprocessors, at least in a MTC framework, since every core into one multiprocessor shares the same control component. As consequence, a greater number of small multiprocessors (TESLA) allows us to reach a better performance than a lower number of big multiprocessors (FERMI and KEPLER).

4.2. Intel Xeon Phi. All of our experiments were ran on the MidWay High- Performance Computing Cluster at University of Chicago. Our testing host is an Intel SandyBridge with 16 cores at 2.6 Ghz and 32 GB of RAM. It has 2 Xeon Phis attached to it (Table 4.2). In this subsection, we focused on analyzing the Matrix Multiplication tasks by using the *OpenMP* approach (Figure 3.2), as the *SCIF* implementation is under development and work is being carried out to run some experiments using Sleep and Matrix Multiplication tasks to measure the performance of the framework.

In order to assess the real-world performance of the Xeon Phi [7], we developed a matrix multiplication application to show how well it performed for various task sizes and levels of concurrency. It should be noted that the work performed is exponentially greater than the matrix size, since a naive matrix multiplication
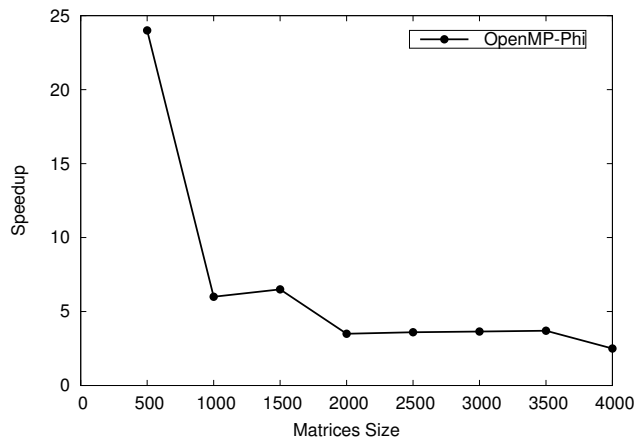
Fig. 4.3: Performance (speedup) achieved by each Intel Xeon Phi by using the *OpenMP* approach.

algorithm of $O(n^3)$ was used.

We used the same speedup used previously for the GPUs analysis, time consumed computing a set of tasks one by one over the time consumed by computing the same set of tasks in parallel at very same time. In particular for our MTC approach and ensuring full utilization of Xeon Phi, we use blocks of 60 tasks. Each of the tasks is mapped on one Phi-core in which we use 4 hardware-threads, $60 \times 4 = 240$ total threads). When one task completes, we send another until 600 tasks are computed. Speedup was calculated by varying the matrix sizes, number of threads and also by varying the level of concurrency of tasks. It was observed (Figure 4.3) that higher performance is achieved with very granular tasks, but the gain reduces as problem scales up to higher matrix sizes. This clearly shows that overhead of data transfer from CPU to MIC is high, which can be mitigated by employing techniques such as allocation a block of memory during the initialization of the framework and reusing the memory blocks for data transfer. Also, performance of sleep jobs was analyzed to assess the ideal performance of Xeon Phi with very short length tasks. It was observed that efficiency in higher 90s could be achieved with tasks lasting as short as 640 microseconds.

5. Conclusions. At the beginning of this work, we described a set of approaches for dealing with MTC over two different many-core architectures, NVIDIA's GPU and Intel Xeon Phi. Also, the main features of both hardware-accelerators were briefly introduced. After that, we analyzed each of the software-hardware approaches individually. In particular, for NVIDIA's GPUs three programming approaches, *Batched*, *Concurrent* and *Dynamic*, were tested on three GPU architectures re-called as TESLA, FERMI and KEPLER. *Batched* and *Concurrent* approaches shown the highest scaling. The overall performance suffers a dramatic fall in performance by increasing the memory requirements and the number of threads, reaching only a good performance over those scenarios with a small demand of memory. Regarding GPUs architecture, we proven that the number of multiprocessor is more relevant that the number of cores to reach a good scaling, at least for our target problem. In this regard, the TESLA architecture (30 multiprocessor and 240 cores) shown a better performance against the KEPLER architecture (13 multiprocessor and 2496 cores). Unlike NVIDIA's GPUs, Intel Xeon Phi turned to be a more appropriate many-core architecture for MTC using an *OpenMP* approach. We obtain a similar trend than obtained in GPUs, the peak performance is reached on very granular tasks, the gain reduces as problem scales up to higher matrix sizes. However, the fall in performance is not so dramatic as in GPUs, and the number of tasks to be efficiently executed is considerable higher than GPUs. Also the peak in performance is much higher, 24 against 9.

REFERENCES

[1] R.J. BARRIENTOS, J.I. GÓMEZ, C. TENLLADO, M.PIEDRO-MATIAS, AND P. ZEZULA, *Multi-level Clustering on Metric Spaces Using a Multi-GPU Platform*, Springer Berlin Heidelberg, Berlin, Heidelberg, 2013, pp. 216–228.

[2] R.J. BARRIENTOS, J. I. GÓMEZ, C. TENLLADO, M. PIEDRO-MATIAS, AND M. MARIN, *knn query processing in metric spaces using gpus*, in Euro-Par 2011 Parallel Processing, Springer Berlin Heidelberg, 2011, pp. 380–392.

[3] S. CHIEN, *Hand-tuned sgemm on gt200 gpu*, Technical Report, Tsing Hua university, R.O.C. (Taiwan), (2010).

[4] INTEL CORP., *Intel many integrated core symmetric communications interface (scif) user guide*, (2012).

[5] NVIDIA CORP., *Nvidia cuda compute unified device architecture-programming guide, version 5*, (2012).

[6] NVIDIA CORP., *Dynamic parallelism in cuda - nvidia technical report*, (2013).

[7] T. CRAMER, D. SCHMIDL, M. KLEMM, AND D. AN MEY, *Openmp programming on intel xeon phi coprocessors: An early performance comparison*, in Proceedings of the Many-core Applications Research Community (MARC) Symposium at RWTH Aachen University, November 2012, pp. 38–44.

[8] S. DIMITROPOULOS, *Gemtc-scif source code repository, https://github.com/sdimitro/scif-modules/tree/ master/scif-sc*.

[9] J. DUATO, J. PENA, F. SILLA, R. MAYO, AND E. S. QUINTANA-ORTI, *Performance of cuda virtualized remote gpus in high performance cluster*, the 40st International Conference on Parallel Processing (ICPP), (2011), pp. 365–374.

[10] H. ESMAEILZADEH, E. BLEM, R. ST. AMANT, K. SANKARALINGAM, AND D. BURGER, *Dark silicon and the end of multicore scaling*, in Proceedings of the 38th Annual International Symposium on Computer Architecture, ISCA '11, New York, NY, USA, 2011, ACM, pp. 365–376.

[11] D. GEER, *Industry trends: Chip makers turn to multicore processors*, Computer, 38 (2005), pp. 11–13.

[12] J. GÓMEZ-LUNA, J.M. GONZÁLEZ-LINARES, J. I. BENAVIDES, AND N. GUIL, *Performance models for cuda streams on nvidia geforce series*, J. Parallel Distrib. Comput., 72 (2012), pp. 1117–1126.

[13] C. GREGG, J. DORN, K. HAZELWOOD, AND K. SKADRON, *Fine-grained resource sharing for concurrent gpgpu kernels*, In Proceedings of the 4th USENIX Workshop on Hot Topics in Parallelism (HotPar), (2012).

[14] M. A. GUEVERA, C. GREGG, K. HAZELWOOD, AND K. SKADRON, *Enabling task parallelism in the cuda scheduler*, In Proceedings of the Workshop on Programming Models for Emerging Architectures (PMEA), in conjunction with the ACM/IEEE/IFIP International Conference on Parallel Architectures and Compilation Techniques (PACT), (2009).

[15] A. HAIDAR, T. DONG, P. LUSZCZEK, S. TOMOV, AND J.J. DONGARRA, *Optimization for performance and energy for batched matrix computations on gpus*, in Proceedings of the 8th Workshop on General Purpose Processing using GPUs, GPGPU@PPoPP 2015, San Francisco, CA, USA, February 7, 2015, 2015, pp. 59–69.

[16] A. HAIDAR, T. DONG, P. LUSZCZEK, S. TOMOV, AND J. J. DONGARRA, *Batched matrix computations on hardware accelerators based on gpus*, IJHPCA, 29 (2015), pp. 193–208.

[17] ———, *Towards batched linear solvers on accelerated hardware platforms*, in Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2015, San Francisco, CA, USA, February 7-11, 2015, 2015, pp. 261–262.

[18] J. JEFFERS AND J. REINDERS, *Intel Xeon Phi Coprocessor High Performance Programming*, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st ed., 2013.

[19] V. J. JIMÉNEZ, LL. VILANOVA, I. GELADO, G. FURSIN M. GIL, AND N. NAVARRO, *Predictive runtime code scheduling for heterogeneous architectures*, the 4th International Conference on High Performance Embedded Architectures and Compilers (HiPEAC), (2009), pp. 19–33.

[20] JEFFREY JOHNSON, SCOTT J KRIEDER, BENJAMIN GRIMMER, JUSTIN M WOZNIAK, MICHAEL WILDE, AND IOAN RAICU, *Understanding the costs of many-task computing workloads on intel xeon phi coprocessors*, 2nd Greater Chicago Area System Research Workshop (GCASR), (2013).

[21] S. KATO, K. LAKSHMANAN, R. RAJKUMAR, AND Y. ISHIKAWA, *Timegraph: Gpu scheduling for real-time multi-tasking environments*, In Proceedings of the 2011 USENIX Annual Technical Conference (USENIX ATC'11), (2011).

[22] C. S. KOUZINOPOULOS, P. D. MICHAILIDIS, AND K. G. MARGARITIS, *Multiple string matching on a GPU using cudas*, Scalable Computing: Practice and Experience, 16 (2015).

[23] J. KREUTZ, *Dgemm-tiled matrix multiplication with cuda*, Jülich Forchungszentrum, (2013).

[24] S. J. KRIEDER, J.M. WOZNIAK, T. ARMSTRONG, M. WILDEL, D. S. KATZ, B. GRIMMER, I.T. FOSTER, AND I. RAICU, *Design and evaluation of the gemtc framework for gpu-enabled many-task computing*, in Proceedings of the 23rd International Symposium on High-performance Parallel and Distributed Computing, HPDC '14, New York, NY, USA, 2014, ACM, pp. 153–164.

[25] J. LIMA, T. GAUTIER, N. MAILLARD, AND V. DANJEAN, *Exploiting concurrent gpu operations for efficient work stealing on multi-gpus*, 24rd International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD), (2012), pp. 75–82.

[26] S. NAKAGAWA, F. INO, AND K. HAGIHARA, *A middleware for efficient stream processing in cuda*, Computer Science - Research and Development, 16 (2010), pp. 197–204.

[27] P. NOOKALA, S. DIMITROPOULOS, K. STOUGH, AND I. RAICU, *Evaluating the support of mtc applications on intel xeon phi many-core accelerators*, in International Conference on Cluster Computing, CLUSTER '15, 2015.

[28] P. Nookala and K. Stough, *Gemtc-openmp source code repository, https://github.com/pnookala/ mic openmp gemtc.*

[29] nVidia Corp., *Just the facts*, Nvidia. Retrieved, (2013).

[30] B. O'Hallaron, *Using blocking to increase temporal locality, http://csapp.cs.cmu.edu/2e/waside/wasideblocking.pdf*, (2013).

[31] GPGPU. General purpose computation using graphics hardware, *http://www.gpgpu.org*.

[32] D. Schmidl, T. Cramer, S. Wienke, C. Terboven, and M. S. Müller, *Assessing the performance of openmp programs on the intel xeon phi*, in Euro-Par 2013 Parallel Processing, Felix Wolf, Bernd Mohr, and Dieter an Mey, eds., vol. 8097 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2013, pp. 547–558.

[33] L. Seiler, D. Carmean, E. Sprangle, T. Forsyth, M. Abrash, P. Dubey, S. Junkins, A. Lake, J. Sugerman, R. Cavin, R. Espasa, E. Grochowski, T. Juan, and P. Hanrahan, *Larrabee: A many-core x86 architecture for visual computing*, in ACM SIGGRAPH 2008 Papers, SIGGRAPH '08, New York, NY, USA, 2008, ACM, pp. 18:1–18:15.

[34] M. Själander, M. Martonosi, and S. Kaxiras, *Power-Efficient Computer Architectures: Recent Advances*, Synthesis Lectures on Computer Architecture, Morgan and Claypool Publishers, Dec. 2014.

[35] R. Smith, *Intel's knights landing xeon phi coprocessor detailed*, June 2014.

[36] TOP500.org, *TOP500 List June 2015*.

[37] P. Valero, J.L. Sánchez, D. Cazorla, and E. Arias, *A gpu-based implementation of the mrf algorithm in itk package*, The Journal of Supercomputing, 58 (2011), pp. 403–410.

[38] P. Valero-Lara, *A gpu approach for accelerating 3d deformable registration (dartel) on brain biomedical images*, in Proceedings of the 20th European MPI Users' Group Meeting, EuroMPI '13, New York, NY, USA, 2013, ACM, pp. 187–192.

[39] P. Valero-Lara, *Multi-gpu acceleration of dartel (early detection of alzheimer)*, in Cluster Computing (CLUSTER), 2014 IEEE International Conference on, Sept 2014, pp. 346–354.

[40] P. Valero-Lara and F.L. Pelayo, *Full-overlapped concurrent kernels*, in Architecture of Computing Systems. Proceedings, ARCS 2015-The 28th International Conference on, VDE, 2015, pp. 1–8.

[41] P. Valero-Lara and F. L. Pelayo, *Towards a more efficient use of gpus*, Computational Science and Its Applications (ICCSA) Workshops, (2011), pp. 3–9.

[42] P. Valero-Lara and Fernando L. Pelayo, *Analysis in performance and new model for multiple kernels executions on many-core architectures*, IEEE International Conference on Cognitive Informatics (ICCI*CC), (2013), pp. 189–194.

[43] P. Valero-Lara, A. Pinelli, J. Favier, and M. Piedro-Matias, *Block tridiagonal solvers on heterogeneous architectures*, in Proceedings of the 2012 IEEE 10th International Symposium on Parallel and Distributed Processing with Applications, ISPA '12, Washington, DC, USA, 2012, IEEE Computer Society, pp. 609–616.

[44] P. Valero-Lara, A. Pinelli, and M. Prieto-Matias, *Fast finite difference poisson solvers on heterogeneous architectures*, Computer Physics Communications, 185 (2014), pp. 1265 – 1272.

[45] J. van Oosten, *Introduction to cuda 5.0*, nVidia, (2014).

[46] B. van Werkhoven, J. Maassen, F.J. Seinstra, and H.E. Bal, *Performance models for cpu-gpu data transfers*, CCGRID, (2014), pp. 11–20.

[47] S. Verdoolaege, J. C. Juega, A. Cohen, J. I. Gómez, C. Tenllado, and F. Catthoor, *Polyhedral parallel code generation for cuda*, ACM Trans. Archit. Code Optim., 9 (2013), pp. 54:1–54:23.

[48] S. Yamagiva and L. Sousa, *Design and implementation of a stream-based distributed computing platform using graphics processing units*, 4th Int. Conf. Computing Frontiers (CF'07), (2007), pp. 197–204.

[49] K. Zhang and B. Wu, *Task scheduling greedy heuristic for gpu heterogeneous cluster involving the weights of the processor*, International Symposium o Parallel & Distributed Processing Workshops (IPDPSW), (2012), pp. 1817–1827.