International Journal of Networking and Computing – www.ijnc.org
ISSN 2185-2839 (print) ISSN 2185-2847 (online)
Volume 4, Number 1, pages 131–150, January 2014

Toward a Generic Hybrid CPU-GPU Parallelization of Divide-and-Conquer Algorithms

Alejandro López-Ortiz
David R. Cheriton School of Computer Science, University of Waterloo
200 University Ave West, Waterloo, ON, N2L 3G1, Canada

Alejandro Salinger
Department of Computer Science, Saarland University
Campus, D-66123 Saarbrücken, Germany

and

Robert Suderman
David R. Cheriton School of Computer Science, University of Waterloo
200 University Ave West, Waterloo, ON, N2L 3G1, Canada

Received: July 27, 2013
Revised: October 17, 2013
Accepted: November 29, 2013
Communicated by Akihiro Fujiwara

Abstract

In the last few years, the development of programming languages for general purpose computing on Graphic Processing Units (GPUs) has led to the design and implementation of fast parallel algorithms for this architecture for a large spectrum of applications. Given the streaming-processing characteristics of GPUs, most practical applications consist of tasks that admit highly data-parallel algorithms. Many problems, however, allow for task-parallel solutions or a combination of task and data-parallel algorithms. For these, a hybrid CPU-GPU parallel algorithm that combines the highly parallel stream-processing power of GPUs with the higher scalar power of multi-cores is likely to be superior. In this paper we describe a generic translation of any recursive sequential implementation of a divide-and-conquer algorithm into an implementation that benefits from running in parallel in both multi-cores and GPUs. This translation is generic in the sense that it requires little knowledge of the particular algorithm. We then present a schedule and work division scheme that adapts to the characteristics of each algorithm and the underlying architecture, efficiently balancing the workload between GPU and CPU. Our experiments show a 4.5x speedup over a single core recursive implementation, while demonstrating the accuracy and practicality of the approach.

Keywords: multi-core, GPU, heterogeneous architectures, hybrid algorithms, performance modeling, divide-and-conquer, parallel algorithms

[0]A preliminary version of this paper appeared in the 15th Workshop on Advances in Parallel and Distributed Computational Models (APDCM 2013) [18].

1 Introduction

Since the appearance of multi-core architectures we have witnessed an increase in algorithms and applications designed to take advantage of the parallel processing capabilities of these now ubiquitous processors. At the same time, there exists a vast collection of graphic applications for optimized performance on Graphic Processing Units (GPUs). Originally designed as specialized processors for graphic operations, the development of accessible programming languages such as CUDA and OpenCL has enabled the use of GPUs for general purpose programming, known as General Purpose computing on GPUs (GPGPU). Consequently, researchers and practitioners have developed algorithms for this architecture for various classes of problems, most notably for problems that admit data-parallel algorithms, many of which fall under the category of the so-called *embarrassingly parallel* problems. In the last few years, the increasing power and the low cost of GPUs have transformed common computers into heterogeneous architectures with tremendous computing power. While for most applications capable of parallel execution there exist implementations either for multi-cores or GPUs, the vast majority of existing applications and algorithms do not yet take full advantage of the available computing power, thus leaving the current processing resources of even middle-end commodity computers largely underutilized. This scenario has motivated the development of projects in several areas, from the proposal of new operating systems [9, 21] to the continuous development of tools and languages to enable an easy transition from traditional CPU code to heterogeneous platforms [28, 19, 20, 27].

Since the vector processing nature of GPUs is suitable for problems that allow efficient data-parallel algorithms, many of the existing GPU algorithms fall into this category. However, many problems allow for parallel algorithms that are task-parallel, or a combination of both task-parallel and data-parallel [12]. Thus, such problems can benefit from the use of CPU cores for non-data-parallel tasks. In fact, plans for the convergence of CPUs and GPUs into one platform by the largest microprocessor manufacturers are becoming a reality [26, 4], which confirms the relevance of algorithms that can be sped-up using the power of both architectures together [8]. Many algorithms for hybrid CPU-GPU architectures have been designed in the last years, most notably for fundamental linear algebra problems [31, 15, 14], among many others. The design of efficient hybrid algorithms encompasses many challenges. A careful task division must be done so that each portion of the algorithm can run on the platform that suits best its characteristics. In addition, it is desirable that algorithms and schedulers adapt to the characteristics and current availability of the computing devices.

In this work we describe a generic approach to develop algorithms for a hybrid CPU-GPU architecture, which we term Hybrid Processing Unit or HPU. We focus on algorithms for a large class of problems suitable for divide-and-conquer solutions. Starting from a sequential recursive implementation of a divide-and-conquer algorithm, we translate this implementation to parallel code that is suitable for running on both CPU and GPU, with a generic translation that can be applied with little knowledge of the particular algorithm. We propose a model for the HPU platform and analyze the optimal division of work for parallel divide-and-conquer under this model. While the analysis presented applies to divide-and-conquer algorithms, the ideas behind it are applicable to other classes of algorithms with structured dependencies between a large number of independent tasks. We then present a case study for the application of our framework using mergesort as a sample algorithm. The simplicity of our implementations confirms the practicality of our approach, while at the same time leading to significant improvements in performance over sequential implementations.

2 Related Work

Several researchers have developed algorithms for heterogeneous architectures. An important set of hybrid algorithm implementations are grouped in the Matrix Algebra on GPU and Multicore Architectures (MAGMA) library [30, 31]. MAGMA provides hybrid implementations of several linear algebra algorithms to enable execution in both multi-core and GPUs, thus extending and adapting the LAPACK [5] and ScaLAPACK [10] libraries to heterogeneous architectures. Examples of these algorithms are Cholesky [3], LU [1, 7], and QR factorizations [2, 17], as well as Hessenberg reductions [32]. In general, the approach for implementing hybrid CPU-GPU code in MAGMA is to

schedule tasks in each computing unit according to their nature: tasks which exhibit small parallelism and that are often on the critical path are scheduled on the CPU while sets of independent tasks are scheduled in the GPU [30, 31]. A recent work implements a hybrid divide-and-conquer strategy for dense symmetric and Hermitian eigenproblems [34]. The divide-and-conquer approach is specific to these problems, in contrast to our generic approach.

Another library of high-performance linear algebra CPU-GPU hybrid implementations is CULA [15], which divides computation so as to enable the CPU and GPU to execute the tasks that each is best suited for, while at the same time carefully overlapping operations in both units.

Hybrid algorithms have been recently developed for other types of problems such as ray tracing [11], encryption and decryption of block cyphers [8], and stencil computations [33], as well as to accelerate domain decomposition methods [23].

An important part of computation on heterogeneous architectures is the proper load balancing among computing units. While some implementations follow analytically determined static schedules [8], others rely on dynamic schedules by runtime systems. StarPU was proposed as a runtime layer to facilitate the dynamic scheduling of parallel tasks in heterogeneous architectures [6]. The programmer is responsible for the implementation of tasks for each computing unit and to declare data dependencies between them, while the runtime system is responsible for handling data movements and efficient scheduling of tasks. The programmer can provide hints for the latter, and thus StarPU provides a high-level framework for the design of scheduling policies. StarPU has been incorporated into MAGMA for dynamic scheduling of routines on multi-GPU environments [1, 2]. Other runtime systems for heterogeneous CPU-GPU architectures are Anthill [29] and SuperMatrix [24].

In this work we target problems with known dependencies, and thus a tailored static division of work suits our purpose. In addition, unlike most of the works described above, which target High Performance Computing applications and solutions, the primary focus of our framework is on generality and ease of programming, and secondarily on performance. While in many cases it is possible to design and implement parallel algorithms for specific problems that will likely exhibit better performance than the general solutions provided by our approach, we emphasize that our main goal is to provide a simple strategy to develop algorithms that can take advantage of the computing power available in commodity computers, rather than extracting the last ounce of performance from a given hardware architecture.

3 A Hybrid CPU-GPU Model

We propose a hybrid CPU-GPU model which we term Hybrid Processing Unit (HPU) and describe a balancing scheme which shares the load optimally in a near automatic fashion for certain well known families of problems. We first briefly review the basic characteristics of GPU architectures. Since we use AMD GPUs in our experimental setup, we describe the GPU architecture from the point of view of the OpenCL standard. The model and ideas presented hereafter, however, are generic and apply to other languages and vendors as well.

3.1 OpenCL Architecture and Programming Model

An OpenCL *platform* consists of a *host* connected to one or more *devices* [16]. Usually, the host corresponds to a CPU and a device to a GPU. An OpenCL device consists of one or more *computing units*, which are further divided into *processing elements (PEs)*. The PEs within the same computing unit execute in Single-Instruction-Multiple-Data (SIMD) mode. An OpenCL program consists of host programs which manage the execution of *kernels* on the device. An instance of a kernel is known as a *work-item*. All work-items execute the same code but the execution path they follow and the data on which they operate might differ across work-items. Each work-item is uniquely identified by a *globalID*, which can be obtained by the item to enable a specific execution path. Work-items are further organized in work-groups, with all work-items in a given work-group executing concurrently in a single computing unit.

An OpenCL device contains different memory regions. Each work-item has access to a *private memory* which is not visible to other work-items. Work-items can share variables in *local memory*.

A *global memory* allows access by work-items in all work-groups. A region of this global memory is known as the *constant memory*, and it is used by the host to allocate memory objects in the device. Data transfers between the host and the device and within different types of memories must be explicitly managed by the program. The performance of a GPU application is highly dependent on its memory access patterns. In general, accesses to local memory are much faster than accesses to global memory. In particular, programs should seek memory accesses to contiguous data by work-items in the same group, which are known as *coalesced* memory operations.

3.2 Hybrid Processing Unit

The HPU consists of a multi-core CPU processor with p cores and a GPU device with g processing elements, which for simplicity we call GPU cores. Given that the true parallelism provided by a GPU varies significantly depending on execution aspects such as scheduling and memory accesses, we do not think about the number of GPU cores as exactly matching the number of physical processing elements but rather as a measure of the empiric degree of parallelism observed when running a suitable test program[1]. In turn, in general the number of CPU cores p might not be equal to the number of physical cores but to a parameter indicating the number of cores available for processing tasks (e.g., one or more cores can handle thread launching or other scheduling tasks).

To account for the different characteristics (and in particular speed) of CPU and GPU cores, we denote as γ_c and γ_g the number of operations per unit of time that a single CPU and GPU core can complete, respectively, with $\gamma_c > \gamma_g$. For simplicity, we normalize these factors, setting $\gamma_c = 1$ and $\gamma_g = \gamma < 1$. We assume that these architectures are balanced in the sense that the ratio γ of operations per unit hold for any kind of operations (logic or memory access), similarly to what is assumed, for example, in [25]. We also assume that $\gamma g > p$, and thus the raw computational power of the GPU is higher than that of the CPU.

The focus in this work is on the most common scenario of one multi-core CPU unit along with one GPU card (which we call *processing units*), although the model could easily be extended to the case of multiple GPU cards.

In terms of communication, transmitting w words between CPU and GPU takes time $\lambda + \delta w$, where λ is a fixed latency cost, and δ is the variable cost per word. For the kind of application that we consider in this work we do not explicitly take this cost into account, but we limit the number of data transfers between processing units to the minimum possible. Similarly, we do not explicitly consider scheduling costs, as preliminary experiments showed that the overhead was negligible.

4 Generic Divide-and-Conquer Parallelization

The standard approach to a divide-and-conquer (DC) algorithm involves dividing the problem into smaller subproblems, recursively solving these subproblems, and combining the solutions of the subproblems into a final solution (see Algorithm 1). We consider DC algorithms whose time complexity can be expressed by:

$$T(n) = aT(n/b) + f(n), \quad T(1) = \Theta(1)$$

Naturally, the algorithm works by dividing the problem in a subproblems of size n/b each and combining their solutions to obtain the final solution to the problem. The division and combination portion of the algorithm takes time $f(n)$. A DC algorithm can be parallelized in a straightforward manner by executing recursive calls in parallel, leading to a simple thread-based implementation suitable for multi-cores. Nevertheless, practical issues such as the efficient use of private and shared caches might hinder the ideal parallel performance of such implementation, with the chosen thread schedule playing a major role in this respect.

In general, a strategy in which each recursive call launches a thread does not suit the GPU multi-processing model, since at least in some architectures GPU threads are unable to launch additional

[1] We defer the details about how to estimate g to Section 6.4.

Algorithm 1 Generic divide-and-conquer implementation

Recursive(param)

```
1: if endCondition(param) then
2:    return BaseCase(param)
3: {param_j} ← Divide(param)
4: for j = 1 to |{param_j}| do
5:    S_j ← Recursive(param_j)
6: S ← Combine({S_j}, param)
7: return S
```

Algorithm 2 Breadth-first divide-and-conquer

BreadthFirst(params)

```
1: split params into basecases and recursions
2: if recursions is empty then
3:    for each param in basecases do
4:       BaseCase(param)
5:    return
6: add basecases to next_params
7: for each param in recursions do
8:    {param_j} ← Divide(param)
9:    for j = 1 to |{param_j}| do
10:      add param_j to next_params
11: BreadthFirst(next_params)
12: for each param in recursions do
13:    Combine(param)
```

threads during execution[2]. Instead, they are designed to run hundreds of parallel threads executing one same kernel launched by the CPU host. In this sense, a breadth first execution of a DC algorithm can suit this execution mode better: the independent tasks on one level of the recursion tree can be seen as the same task being executed on different parts of the input. Given enough independent tasks, one kernel can be launched to execute an entire level of the tree in parallel.

4.1 Breadth-First Structure

The first step of our strategy to obtain a hybrid implementation of a DC algorithm is to convert the sequential code from the form in Algorithm 1 to one that will execute in breadth-first order. We do this by replacing multiple recursive calls with one recursive call that represents multiple subproblems, encoded in the parameters of the recursive call. Algorithm 2 shows the modified pseudocode. At each level of the recursion, parameters for all subproblems are encoded in *params*, some of which correspond to base cases. The rest of the parameters (recursions) are divided according to the algorithm's divide procedure (line 7) and grouped in one list of parameters (next_params) to be passed on to the one recursive call in line 11. After the recursive call, the results of subproblems in each group in the level are combined according to the combine procedure (line 12). Note that at each level subproblems corresponding to base cases are passed on to the next recursive call, and their execution is delayed until no more recursive calls remain.

4.2 Conversion to GPU Code

In order to modify the existing code for GPU execution, the thread launching system and the base-case, division, and combination steps must be suitably adapted. Each subproblem will have a

[2]CUDA 5 does implement dynamic parallelism, enabling kernels to launch other kernels without involvement of the host. However, currently the depth of the nested computation can be at most 24 and is limited by the availability of resources in the GPU [22]. While this might change in the future allowing for a more suitable strategy for divide-and-conquer implementations, our breadth-first strategy remains valid and applicable to a wider range of architectures which are not CUDA-enabled.

Algorithm 3 Pseudo-code for functionGPU

functionGPU(parameters, base)

```
1: id ← get_global_id()
2: param ← parameters[id]
3: memory = base + fn(id, param)
4: thread_function(param, memory)
```

Algorithm 4 Pseudo-code for Sum

sum(array, size)

```
1: if size > 1 then
2:     sum(array, size/2)
3:     sum(array + size/2, size/2)
4:     array[0] ← array[0] + array[size/2] {array[0] stores the result}
```

separate thread. As GPU threads have a relatively small overhead for launching more threads than available cores, the advantage of processing as many tasks in parallel will take priority over launching only as many threads as can be run in parallel. Then, during execution, each GPU thread is provided a unique thread id that can be used to load its unique set of parameters and to determine any (previously allocated) memory blocks on which it will operate. This generic description is shown in Algorithm 3, where fn is a function on the thread's id and parameters that determines the thread's relevant memory blocks, and **thread_function** denotes the operations performed by the GPU thread.

4.3 Example: Divide-and-Conquer Sum

We show an example of the code translation described above for a simple divide-and-conquer procedure that computes the sum of elements in an array (see Algorithm 4). The pseudocode of the resulting GPU program is shown in Algorithm 5. This program is executed at each level of the recursion, with numSubProblems indicating the number of subproblems at the current level. For a level with b subproblems, the i-th thread computes the sum of elements i and $i+b$ in the array. The final result is stored in $array[0]$. In this case, the relevant parameter to the thread is only numSubProblems, and the relevant memory blocks for the thread are given by its id and numSubProblems. Lines 2-3 in Algorithm 5 correspond to **thread_function** in Algorithm 3.

5 Work Division and Scheduling Strategies

A proper work division and scheduling is key to an efficient implementation. In a heterogeneous architecture, tasks in an algorithm should be assigned to each processing unit according to the tasks' nature and dependencies. Implementations on CPU-GPU architectures generally assign to the CPU tasks with many dependencies [14] or tasks in the critical path [30]. In the case of the regular DC algorithms that we consider in this work, what we regard as *tasks* are the division and combination portions of the algorithm. Hence, all tasks are similar to each other in nature. On the other hand, in regular DC algorithms all paths from the root to the leaves in the tree have approximately equal lengths, and in this sense there are several critical paths, thus they cannot all be assigned to CPU cores. Instead, we assign tasks to the CPU or GPU according to the availability of parallel independent tasks at each level of the recursion tree. Note that although at each level of the recursion there are several subproblems whose division and combination functions are independent and can execute in parallel, the division and combine functions of each subproblem remain sequential. In other words, the only source of parallelism is in the recursive calls, and we do not consider parallelizations of divide and combine functions of particular algorithms. Recall that our main goal is to provide a generic approach for translating recursive DC algorithms to hybrid code with little effort.

It is desirable that the task division involves minimal communication costs. With this goal in

Algorithm 5 Pseudo-code for GPU Sum

sum(numSubProblems, array)

```
1: id ← get_global_id()
2: if id < numSubProblems then
3:     array[id] ← array[id] + array[id + numSubProblems]
```

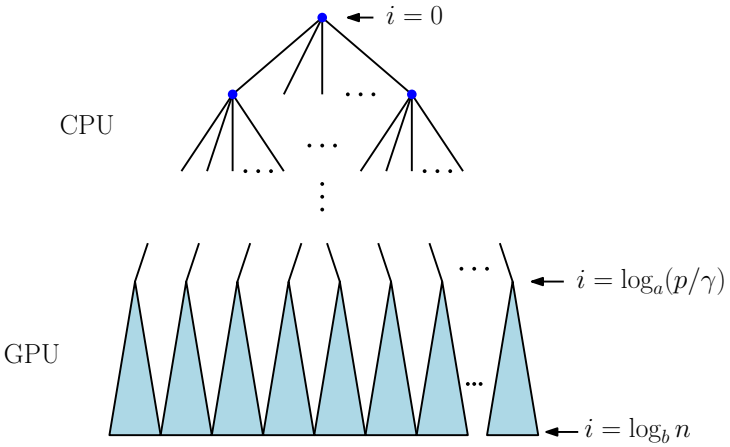

Figure 1: Basic hybrid work division. Each level of the recursion tree is executed in the platform in which it runs faster according to the characteristics of the architectures and the divide-and-conquer problem. Thus, levels above $i = \log_a(p/\gamma)$ are executed in the CPU (while the GPU is idle), whereas levels below i are executed in the GPU (while the CPU is idle).

mind, we design two division and scheduling strategies, which we call basic and advanced. We explain next these strategies and analyze the conditions for scheduling tasks in each computing unit for each strategy.

5.1 Basic Hybrid Work Division

Consider the recursion tree of a DC algorithm as depicted in Figure 1. Each level of the recursion tree consists of division and combination tasks that are independent of each other and can thus be executed in parallel. The basic work division strategy schedules each level on the CPU or GPU depending on where it is more efficient to execute the entire level. There is an advantage to schedule a level in the GPU when the number of independent subproblems allows the use of enough cores to overcome their comparatively slower speed.

Consider the execution time of each processing unit for a given level i in the recursion tree, where $0 \leq i \leq \log_b(n) - 1$, the 0-th level being the top of the tree. Recall that the DC algorithm solves a subproblems of size b and that the division and combination steps take time $f(n)$.

1. $0 \leq i < \log_a(p)$: $T_{CPU}(n,i) = f(n/b^i)$, $T_{GPU}(n,i) = f(n/b^i)/\gamma$. Since $\gamma < 1$, it is faster to run the level on the CPU.

2. $\log_a(p) \leq i < \log_a(g)$: $T_{CPU}(n,i) = (a^i/p)f(n/b^i)$, $T_{GPU}(n,i) = f(n/b^i)/\gamma$. It becomes faster to run the level on the GPU when $a_i/p \geq 1/\gamma$, i.e., $i \geq \log_a(p/\gamma)$.

3. $\log_a(g) \leq i < \log_b(n)$: $T_{CPU}(n,i) = (a^i/p)f(n/b^i)$, $T_{GPU}(n,i) = (a^i/(\gamma g))f(n/b^i)$. Since we assume $g\gamma \geq p$, it is faster to run the level on the GPU.

4. leaves: $T_{CPU}(n) = n^{\log_b a}/p$, $T_{GPU}(n) = n^{\log_b a}/(\gamma g)$. Again it is faster to run the leaves on the GPU.

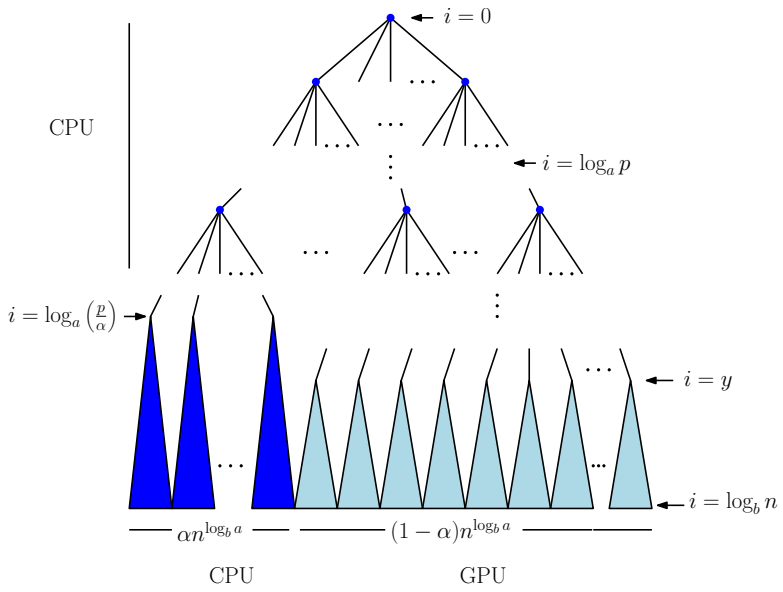

Figure 2: Advanced hybrid work division. The GPU will execute so long as the CPU has enough tasks to keep all cores busy (until it reaches level $\log_a(p/\alpha)$ in a bottom up execution of the left part of the tree in the figure), while keeping only one transfer between processing units. The goal is to find the value of α that maximizes the total work executed by the GPU. In the figure, dark blue and light blue subproblems are executed by the CPU and GPU, respectively.

Hence there is only one transfer at level $i = \log_a(p/\gamma)$. Note that if $g\gamma < p$ then at every level it is faster to execute on the CPU and there is no transfer to the GPU at any point with this strategy. As only a single data synchronization point occurs when work is transferred from the CPU to the GPU and back, transfer time is minimized and accounts for only a small part of the overall processing time.

A drawback of this strategy is that at any point only one of the computing units (i.e., CPU or GPU) is active. We now describe a strategy that builds on this basic strategy while minimizing idle periods.

5.2 Advanced Hybrid Work Division

The new strategy that we propose for dividing the work among CPU and GPU aims to minimize idle periods and communication between units. If all tasks could be executed in parallel the division would be straightforward: we should divide the work so that both processing units take the same time in their assigned portions. However, in general in DC algorithms there is not enough parallelism at all levels of the tree to maintain a uniform division of work. For example, when the number of available subproblems is less than p, any fraction of them assigned to the GPU will leave at least one CPU core idle. Since CPU cores are faster than GPU cores, it is preferable to execute these serial tasks in the CPU. Therefore, all tasks at levels of the recursion tree where there is at most p subproblems ($0 \leq i \leq \log_a p$) are assigned to the CPU (see Figure 2).

For lower level of the trees, we can execute some tasks on the GPU while keeping all CPU cores busy. Consider the recursion tree depicted in Figure 2. When there are more than p subproblems, there is an advantage in running some subproblems in the CPU and some in the GPU. The idea is to run subproblems in the GPU so long as no CPU core is idle. At the same time, we want to keep communication and synchronization costs low. Toward this end, we restrict the number of data transfer between CPU and GPU to two points during the execution. Let y be a parameter denoting the level in the tree (from the top) at which we offload any computation to the GPU, and let α be the parameter denoting the fraction of subproblems that are assigned to the CPU. Thus, at level y,

the CPU is assigned αa^y subproblems, while the GPU is assigned $(1-\alpha)a^y$. Note that once a level has been divided in this way, lower levels of the tree will keep the same fraction of subproblems for each processing unit, and hence no further synchronization or data transfer is required until the GPU has solved all subproblems. The choice of y and α determines the amount of work that the GPU will do. Our strategy maximizes the work that the GPU does with two data transfers while avoiding idle CPU cores. In the rest of this section we show how to determine the parameters y and α that maximize GPU work.

5.2.1 Parameter Optimization

For the sake of analysis, consider a bottom up execution of the recursion tree in Figure 2. Starting from the bottom level, both CPU and GPU execute with a work ratio of α. Since we want to avoid idle CPU cores, we run both CPU and GPU until the CPU portion reaches p subproblems. We only consider $\alpha \geq p/n$, so that the CPU starts at the bottom level with at least p tasks. Then, the CPU portion reaches p subproblems at level $\log_a(p/\alpha)$. At this point, we stop the GPU execution and let the CPU compute all the unfinished portions of the tree. The time that it takes to the CPU to reach level $\log_a(p/\alpha)$ from the bottom is

$$T_c(n) = \frac{\alpha}{p}\left(n^{\log_b a} + \sum_{i=\log_a(p/\alpha)}^{\log_b(n)-1} a^i f\left(\frac{n}{b^i}\right)\right)$$

During this time, the GPU executes bottom up and reaches level y in the tree. The value of y can be determined by making the GPU and CPU times equal. We have 3 cases, depending on whether the GPU is never saturated or always saturated throughout, or a combination of both. Let $T_g^{max}(n)$ denote the maximum time the GPU can execute while using all its cores (i.e., before it reaches g subproblems). Thus,

$$T_g^{max}(n) = \frac{(1-\alpha)}{\gamma g}\left(n^{\log_b a} + \sum_{i=\log_a(g/(1-\alpha))}^{\log_b(n)-1} a^i f\left(\frac{n}{b^i}\right)\right)$$

(i) $(1-\alpha)n^{\log_b a} < g$. In this case, the GPU is never saturated and thus $T_g(n) = (1/\gamma)(1 + \sum_{i=y}^{\log_b(n)-1} f(n/b^i))$.

(ii) $T_c(n) \leq T_g^{max}(n)$. In this case, the GPU is always saturated and thus $T_g(n) = \frac{1-\alpha}{\gamma g}(n^{\log_b a} + \sum_{i=y}^{\log_b(n)-1} a^i f(n/b^i))$.

(iii) $T_c(n) > T_g^{max}(n)$. In this case, $T_g(n) = T_g^{max}(n) + (1/\gamma)\sum_{i=y}^{\log_a(g/(1-\alpha))-1} f(n/b^i)$.

The goal is to determine the value of α that maximizes the work done by the GPU from the bottom until level y. We first determine y by solving the equation $T_g(n) = T_c(n)$ for each of the 3 cases above. This yields a piecewise function $y = y(\alpha)$. The work done by the GPU in this period is given by

$$W_g(n) = (1-\alpha)\left(n^{\log_b a} + \sum_{i=y(\alpha)}^{\log_b(n)-1} a^i f\left(\frac{n}{b^i}\right)\right).$$

Maximizing for α yields the optimal work ratio value.

After the GPU reaches level y, it transfers the results back to the CPU, which finishes the computation. Note that it could still be advantageous to continue execution on the GPU for levels above y. However, this would invariably imply either having idle CPU cores or a new work ratio α, which would in turn imply further synchronization and data transfer between processing units.

Algorithm 6 Pseudo-code for Mergesort

```
mergesort(array, size)
1: if size > 1 then
2:     mergesort(array, size/2)
3:     mergesort(array + size/2, size/2)
4:     merge(array, array + size/2, size/2)
```

5.2.2 Example

We illustrate this procedure using the characteristics of a sample architecture and a divide-and-conquer algorithm whose division and combination function takes time $\Theta(n^{\log_b a})$ (and thus $T(n) = \Theta(n^{\log_b a} \log n)$). Mergesort is an example of such algorithm. We assume that the implementation of the combination and division function is the same both in the CPU and GPU, and thus the constants hidden in the complexities are the same and will cancel out when solving for the level y.

The time that the CPU takes to reach p problems from the bottom is

$$T_c(n) = \frac{\alpha n^{\log_b a}}{p}\left(\log_b n - \log_a \frac{p}{\alpha} + 1\right).$$

The maximum time the GPU can be fully saturated is

$$T_g^{max}(n) = \frac{(1-\alpha) n^{\log_b a}}{\gamma g}\left(\log_b n - \log_a \frac{g}{1-\alpha} + 1\right).$$

Thus, we have the following function for the GPU time:

$$T_g(n) = \begin{cases} (1/\gamma)(n^{\log_b a}\frac{a}{a-1}a^{-y} - \frac{1}{a-1}), & \text{if } (1-\alpha)n^{\log_b a} < g \\ \frac{(1-\alpha)n^{\log_b a}}{\gamma g}(\log_b n - y + 1), & \text{if } (1-\alpha)n^{\log_b a} \geq g \text{ and } T_g^{max}(n) \geq T_c(n) \\ T_g^{max}(n) + n^{\log_b a}\frac{a}{\gamma(a-1)}\left(a^{-y} - \frac{1-\alpha}{g}\right), & \text{if } (1-\alpha)n^{\log_b a} \geq g \text{ and } T_g^{max}(n) < T_c(n) \end{cases}$$

We now solve $T_c(n) = T_g(n)$ for y for each of the cases above, obtaining a piecewise function $y(\alpha)$ that depends on each case. The work done by the GPU is then

$$W_g(n) = (1-\alpha)n^{\log_b a}(\log_b n - y(\alpha) + 1).$$

By replacing in this equation the parameters of a divide-and-conquer algorithm, of a particular architecture, and the input size, we can maximize the work using numeric methods. For example, using mergesort as the divide-and-conquer algorithm and the parameters of one of our architectures[3] (i.e., $a = b = 2$, $f(n) = \Theta(n)$, $p = 4$, $g = 2^{12}$, $\gamma = 1/160$) and an input size $n = 2^{24}$, we obtain the y function and the fraction of GPU work over total work function depicted in Figure 3. In this case the total work is $n^{\log_b a}(\log_b n + 1)$. The work ratio that maximizes the GPU work is $\alpha^* \approx 0.16$, for which the GPU does approximately 52% of the total work. The level reached by the GPU with α^* is approximately 10. Since $\log_2 g = 12$, this means that for the GPU is both saturated and non-saturated during its execution for $\alpha = \alpha^*$. Figure 4 depicts the work division for this example.

6 Case Study: Mergesort

The ideas of our method are applicable in general to algorithms whose parallel structure can be specified by directed acyclic graphs. In this section we use mergesort as a test case for the gains of our general framework for divide-and-conquer algorithms. We particularly chose mergesort as an example of a task-parallel algorithm that is not readily made for execution on a GPU, but that nevertheless is amenable to the kind of hybrid parallelization that we propose.

[3]The architectures and parameters are described in Section 6.4.

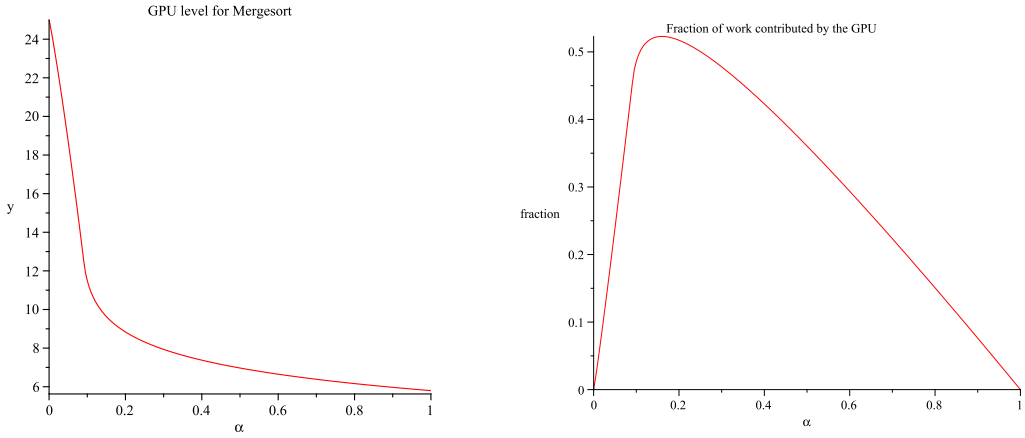

Figure 3: For mergesort ($a = b = 2$, $f(n) = \Theta(n)$) and parameters $p = 4$, $g = 2^{12}$, $\gamma^{-1} = 160$ and $n = 2^{24}$, (left) level reached by the GPU while the CPU has at least p tasks at the same level as a function of the work ratio α, and (right) the percentage of work done by the GPU as a function of α.

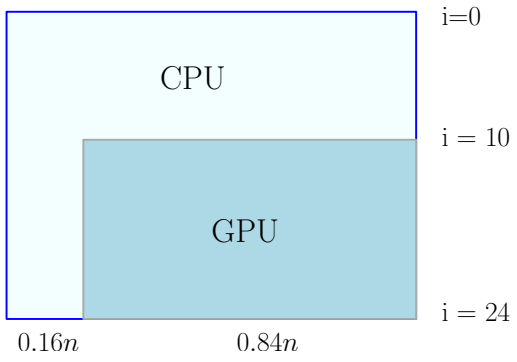

Figure 4: Advanced hybrid work division for mergesort. The figure represents the recursion tree shown in Figure 2 with the height and width of the rectangles representing the height and work of the computation. For the parameters in the example ($p = 4$, $g = 2^{12}$, $\gamma^{-1} = 160$ and $n = 2^{24}$) the work ratio that maximizes the GPU work is $\alpha \approx 0.16$ and the transfer level is 10.

Algorithm 7 Pseudo-code for Breadth-first Mergesort

```
mergesort_bf(array, totalSize, size, numSublists)
1: if size > 1 then
2:    mergesort_bf(array, totalSize, size/2, 2 · numSublists)
3:    for i = 0 to numSublists − 1 do
4:       offset ← i · size
5:       merge(array + offset, array + offset + size/2, size)
```

Consider the classic recursive mergesort implementation as shown in Algorithm 6. As described in Section 4.1, we first convert the recursive divide-and-conquer implementation to a breadth-first one. Compared to the pseudocode in Algorithm 2, a breadth-first execution of mergesort is somewhat simplified as the division into subproblems and condition for basecases are data independent. Thus, a single recursion is performed with parameters indicating the sublists to be sorted. A sublist can be specified by an offset with respect to the beginning of the entire list and the size of the sublist. The offset for the i-th sublist is simply $offset = i \cdot size$. This limits the parameters to the array being sorted, the total array size, and the number of sublists, which are the same for each sublist. Furthermore, determining when only base cases remain becomes trivial: once the number of current sublists equals or exceeds the total length of the array, the maximum number of elements in a sublist is one, and therefore only base cases remain. To finish the conversion, the base-case, division, and combination steps must be performed for each sublist. For mergesort, as no division and base case exist, these parts are removed entirely and only the combine step must be performed, which corresponds to merging pair of sublists. Algorithm 7 shows the breadth-first mergesort implementation[4].

6.1 Basic Hybrid Implementation

The merge operations for each sublist in line 5 in Algorithm 7 are independent of each other and can potentially be executed in parallel. For the basic hybrid work division as described in Section 5.1, these operations are executed either on the GPU or on one or more CPU cores. For each recursion level in which merge operations are executed on the GPU, a program running in the host launches a GPU kernel with parameters indicating the size and number of sublists to be merged. Based on its id, each GPU thread identifies the sublist on which it will operate. Similarly, when merge operations are executed on the CPU, depending on the recursion and number of cores available, multiple threads are created to merge sublists in parallel (with each thread merging a pair of sublists sequentially).

6.2 Advanced Hybrid Implementation

The advanced work-division strategy as described in Section 5.2 is implemented by, when reaching certain threshold level in the recursion tree, launching two simultaneous CPU threads, one that executes the basic hybrid strategy, and another one that executes a CPU implementation (see Figure 2). For these threads, the input is divided according to the division ratio α. Since this ratio remains constant across levels, when the hybrid thread switches to execution on the GPU (level y in Figure 2), the number of subproblems executed in each processing unit will respect the chosen ratio. This implementation is shown in Algorithm 8. In this algorithm, the methods **mergesort_bf_cpu** and **mergesort_bf_hybrid** correspond, respectively, to implementations of Algorithm 7 to be executed exclusively on the CPU, and in a hybrid fashion according to the basic model.

6.3 GPU Optimizations

So far, the hybrid implementation of mergesort described above is oblivious to the characteristics of the particular divide and combine functions. In the case of the merge method, certain optimizations

[4]In order to keep the description of the approach simpler, we assume that the input size is a power of 2. The same general approach is applicable in general, although some adjustments to the implementation are required.

Algorithm 8 Pseudo-code for Advanced Hybrid Mergesort

```
HybridMergesort( array, totalSize, size, numSublists)
 1: if size > 1 then
 2:   if numSublists > threshold then
 3:     cpuLists ← α · numSublists
 4:     gpuLists ← numSublists − cpuLists
 5:     mergesort_bf_cpu(array, size · cpuLists, size, cpuLists)
 6:     mergesort_bf_hybrid(array + size · cpuLists, size · gpuLists, size, gpuLists)
 7:   else
 8:     HybridMergesort(array, totalSize, size/2, 2 · numSublists)
 9:   for i = 0 to numSublists − 1 do
10:     offset ← i · size
11:     merge(array + offset, array + offset + size/2, size)
```

Table 1: Specification of hybrid platforms used in experiments.

Platform	CPU	GPU
HPU1	Intel® Core™ 2 Extreme CPU Q6850	ATI Radeon™ HD 5970
HPU2	AMD A6 3650	ATI Radeon™ HD 6530D

to the implementation are possible and have a significant impact on performance. In order to achieve coalesced memory accesses, prior to executing a parallel merge operation on the GPU, we permute the input so that the set of i-th elements in all sublists are in contiguous locations. Thus, various parallel threads operating on different sublists will access contiguous memory segments. To adapt the GPU kernel to use this method, sublists are iterated using the thread id as the initial position, and increasing this value by the total number of sublists. As the CPU cache benefits from reading from sequential blocks, before transferring the array to the CPU, the array is permuted back to the original arrangement. Thus this optimization is transparent to the CPU implementation. We note that by incorporating this optimization, which is specific to the application, we have chosen to improve performance at the cost of some minor generality. Similar considerations could be applied to other applications, and one can choose whether to incorporate them or not, depending on their difficulty of implementation and the performance gains they may lead to.

6.4 Experimental Results

We implemented the hybrid mergesort algorithm and tested its performance on two OpenCL platforms: an Intel® Core™ 2 Extreme CPU Q6850 (4 cores at 3.00 GHz, 8 Mb cache) with an ATI Radeon™ HD5970[5] GPU card (which we call HPU1), and an AMD Accelerated Processing Unit A6 3650 (4 cores at 2.6 GHz, 4 Mb Cache) with an integrated ATI Radeon™ HD 6530D card (called HPU2) (see Table 6.4). The algorithms were implemented with OpenCL 1.1 AT-Stream-v2.3 in Ubuntu 10.04.4 64-bit (HPU1), and OpenCL 1.2 AMD-APP in Ubuntu 12.04 64-bit (HPU2).

We estimated the parameters γ (ratio between CPU and GPU scalar performance) and g (number of GPU cores) for each platform as shown in Table 6.4. Recall that g does not actually correspond to the physical number of cores or processing elements of the GPU but rather to an approximation of the number of threads that fully saturates the device when running a suitable procedure. In this case, in order to estimate g, we ran an implementation of an elementwise sum of two arrays in which all threads worked in consecutive array segments. We measured the running time as the number of threads used increased, and set g to the value after which no improvement in performance

[5]The HD5970 is a Dual GPU card, but only one card was used in the experiments, as the parallelism available in the application could only saturate both cards at the lowest levels of the recursion tree, not justifying the overhead of additional data transfers.

Table 2: Platforms parameters (p: number of CPU cores, g: number of GPU cores, γ: CPU-GPU scalar performance ratio).

Platform	p	g	γ^{-1}
HPU1	4	4096	160
HPU2	4	1200	65

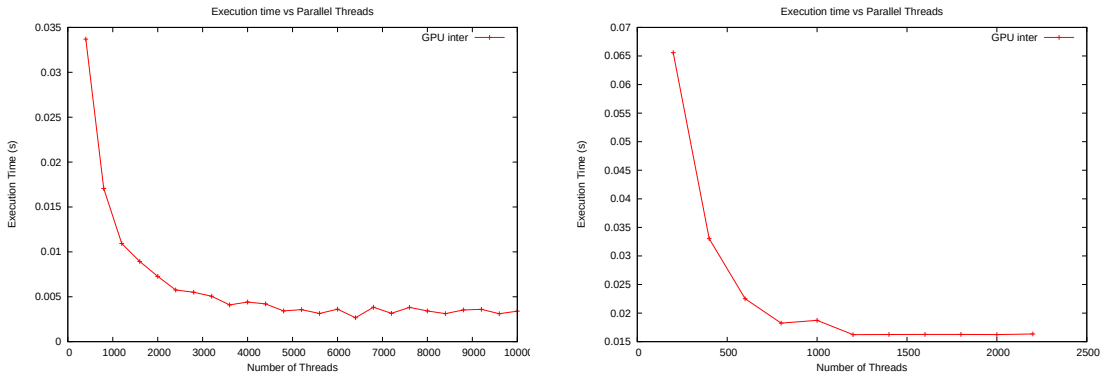

Figure 5: Running time as a function of the number of GPU threads used in an elementwise sum of two arrays for platform HPU1 (left) and HPU2 (right). The size of each array is 2^{24}.

was detected[6]. Figure 5 shows the running times as a function of the number of threads for each platform. The parameters were set to $g = 4096$ for HPU1 and $g = 1200$ for HPU2. In order to estimate γ, a 1-thread merge operation over two lists was executed on both CPU and GPU. Figure 6 shows the running time for different input sizes. As expected, the time ratio remains relatively constant. These parameters were set to $\gamma^{-1} = 160$ for HPU1 and $\gamma^{-1} = 65$ for HPU2.

We measured the performance of the advanced hybrid mergesort implementations for various transfer levels and ratios. Figure 7 shows the speedups of the hybrid implementation on HPU1 (using 4-CPU cores) with respect to a 1-core CPU recursive implementation, as a function of the ratio α for various transfer levels for an input of size $n = 2^{24}$ (elements in all input sequences were chosen uniformly at random between 0 and $2n - 1$). Recall from the example in Section 5.2.2 that for this input size the estimated optimal ratio and transfer levels were $\alpha \approx 0.16$ and $y = 10$. We observe that the speedups do not differ too much across transfer levels, although speedups increase from level 7 and start decreasing with level 11, in accordance with the estimation. Similarly, the performance is slightly better for transfer ratios that are close to the estimated one.

Figure 8 shows the speedups obtained in both platforms with the values of transfer level and ratio that resulted in the best speedups (in red). The green lines in the figures show the speedup estimated in the advanced model analysis for the parameters of the platforms. The maximum speedups achieved were 4.54x for HPU1 and 4.35x for HPU2, which are close to the estimated 5.47x and 5.7x by the analysis for the corresponding input size, respectively. Recall that the overall gains in performance are limited by the sequential execution of the merge methods on large input sizes at the top levels of the recursion tree, which also limits the performance of a similar multi-core only implementation to 2.5x-3x speedups on 4 cores [13]. In this sense, the performance gains obtained through the hybrid implementation are considerable, as we should take into account that according to the analysis the GPU does about 50% of the total work, which in the best case could lead to a

[6]The sum of two arrays used to estimate g shares the characteristics of the merge with optimization for memory coalesced access of our application, and thus it provides a good approximation of the degree of parallelism of the architecture in this case. Another option would have been to use the same merge function, and in general, the same divide or combine function of the application can be used for this purpose. Note that the parameter estimation is done only once.

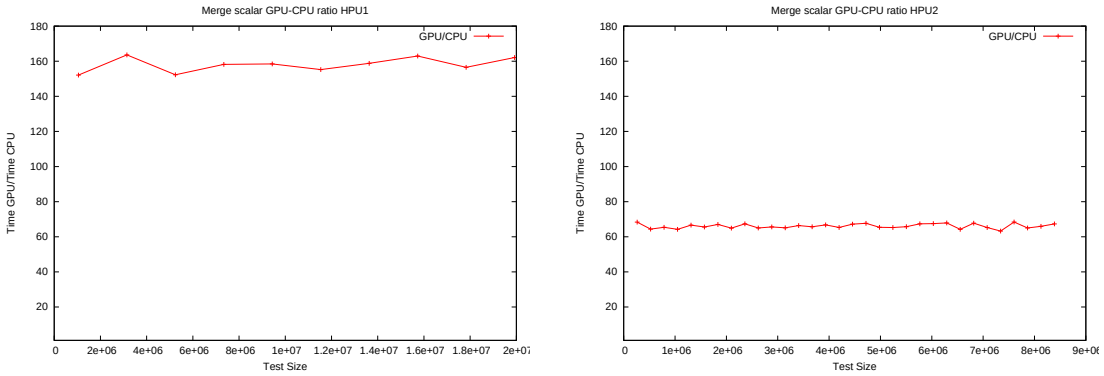

Figure 6: Ratio between scalar performance of single GPU and CPU cores when executing a merge operation on HPU1 (left) and HPU2 (right).

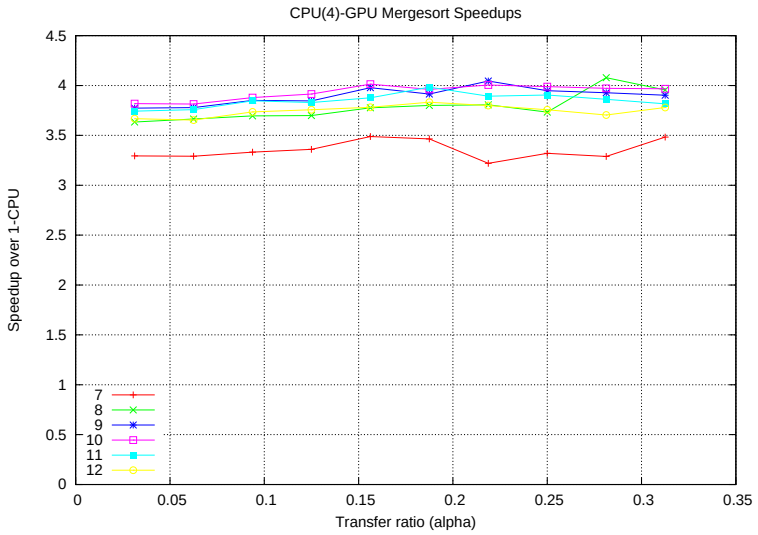

Figure 7: Speedup of hybrid mergesort implementation on HPU1 with an instance of size $n = 2^{24}$ as a function of the transfer ratio α. Each curve corresponds to a different transfer level between processing units (parameter y in Figure 2).

2x speedup over a 4-core execution. Recall as well from the advanced hybrid model description in Section 5.2 that the GPU should execute so long as the CPU has enough tasks to keep cores busy (as shown in the bottom of the tree in Figure 2). The blue line in Figure 8 shows the ratio between these parallel GPU and CPU times. Observe that the ratio is in general close to one and that the best speedup points coincide with the instances in which this ratio is closest to one.

We observe as well that as the input size grows, the obtained speedups (red) decrease and do not keep up with estimated ones (green). We believe that as the input size increases, poor cache utilization hurts the performance of the multi-core portion of the execution. Speedups start to decrease around an input size of $n = 2^{20}$. The space used by the algorithm is roughly $2n \cdot$sizeof(int), i.e., $2^{23} = 8$ Mb. The sizes of the last level CPU caches in HPU1 and HPU2 are 8 Mb and 4 Mb, respectively. Thus, for larger input sizes multiple cores will compete for cache use.

For the sake of comparison with a fully parallel solution, we show the times and speedups of a mergesort GPU implementation that implements a parallel algorithm for the merge phase. Like the implementation with sequential merge, the parallel GPU implementation executes the recursion

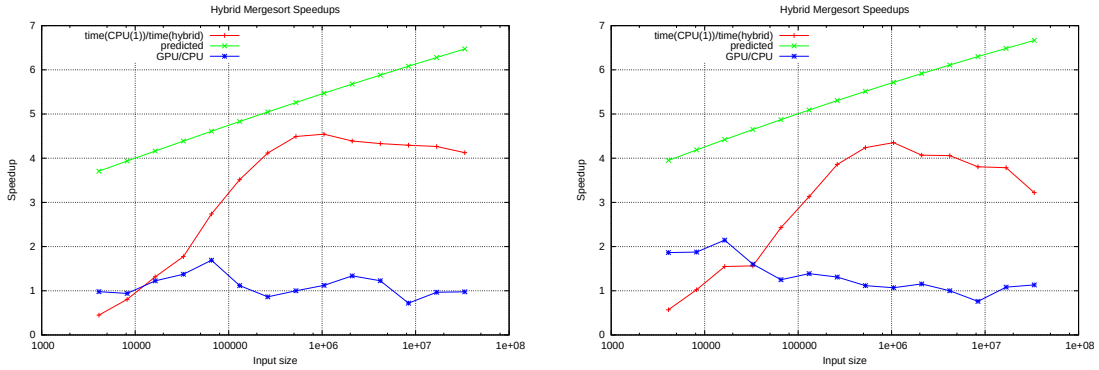

Figure 8: Speedup of hybrid mergesort implementation (red) as a function of the input size for HPU1 (left) and HPU2 (right). The green line depicts the estimated speedups in the analytical model. The blue line shows the ratio between the time of execution of GPU and the time while the CPU is fully utilized (see Section 5.2).

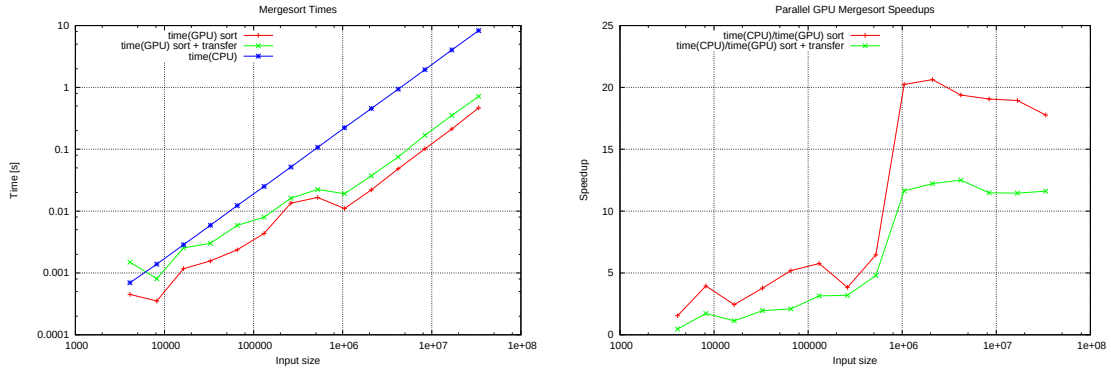

Figure 9: Times (left) and speedups (right) of a GPU only implementation of mergesort with parallel merge compared to a sequential CPU recursive implementation as a function of the input size running on HPU1. Red lines correspond to the times and speedups for sorting only on the GPU while the green lines include the time of data transfers.

tree in breadth-first order as well, merging pairs of sublists in each level. Merging two sublists is implemented by performing a binary search for each element in parallel in order to find its position in the merged list. Figure 9 shows the times and speedups compared to a recursive divide-and-conquer execution on one CPU core on HPU1. We observe that speedups are only significantly larger than those of our solution for large input sizes, reaching 18x-20x speedups for sorting only and being reduced to about 12x when considering the overhead of data transfers.

Finally, to see how the resulting best parameters compare to the predicted ones by the advanced hybrid model, Figure 10 shows the ratio α and transfer level y that resulted in the smallest running times for each input size compared to the ones predicted by the model for HPU1. Note that resulting parameter values are closer to the predicted ones as the input size grows, which coincides with higher speedups. Observe as well that in the case of the optimal transfer level, the obtained values essentially coincide with the predicted ones for larger values of the input size, as the fractional numbers shown in the figure can only take integer values in an actual execution.

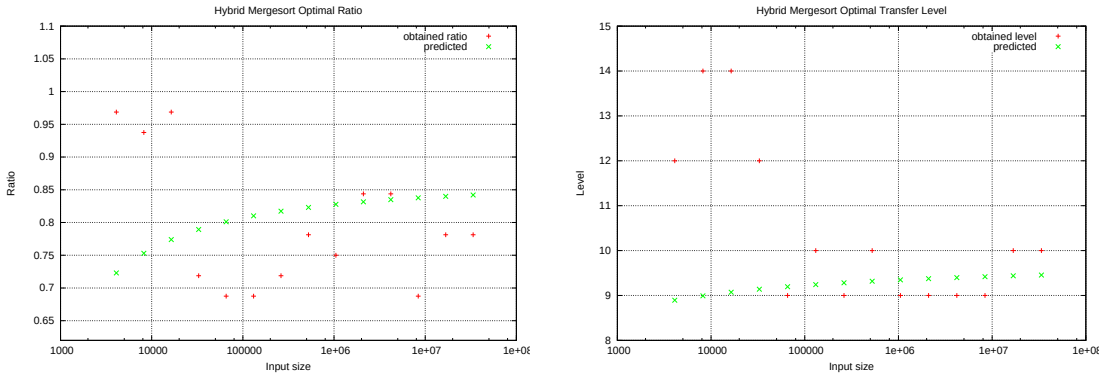

Figure 10: Red points show the work ratio α (left) and transfer levels y (right) between CPU and GPU that resulted in the smallest running times for each input size (for HPU1). Green points correspond to the optimal values as predicted by the model.

7 Conclusions

We presented the Hybrid Processing Unit, a model for hybrid computation on heterogeneous CPU-GPU architectures. In this model, we describe a generic framework to implement hybrid divide-and-conquer algorithms and provide a work-division strategy that minimizes idle times and communication between processing units.

Experimental results on a mergesort example confirm the accuracy of the model at predicting speedups and parameters that yield the best performance, thus suggesting that a model based on traditional approaches to the design and analysis of parallel computation can be useful in a heterogeneous scenario.

For future work, it would be interesting to explore optimizations to the scheduler of the model to obtain better performance for some problems. For example, for problems in which the parallelization of the divide and conquer portions of algorithms is simple—such as dense matrix operations—, the recursive schedule could be stopped at a certain level of the tree, after which parallel versions of the GPU kernels could be executed. Another approach that could lead to performance gains could be to switch to non-recursive sequential versions of the algorithms at the lowest levels of the tree. In either case, the optimal switching level and CPU-GPU work ratio would have to be determined either analytically or experimentally for each particular application.

In addition, we plan to refine the model by considering cache, communication, and scheduling costs explicitly, as well as to extend its applicability to other classes of problems that are suitable for obtaining performance gains in heterogeneous architectures.

Acknowledgments

This project was partially funded by the Natural Sciences and Engineering Council of Canada (NSERC) and Advanced Micro Devices, Inc. (AMD) through the NSERC Engage Grants Program. We would like to thank anonymous reviewers for comments that helped to improve the presentation of this paper.

References

[1] E. Agullo, C. Augonnet, J. Dongarra, M. Faverge, J. Langou, H. Ltaief, and S. Tomov. LU factorization for accelerator-based systems. In *Computer Systems and Applications (AICCSA), 2011 9th IEEE/ACS International Conference on*, pages 217 –224, Dec. 2011.

[2] E. Agullo, C. Augonnet, J. Dongarra, M. Faverge, H. Ltaief, S. Thibault, and S. Tomov. QR factorization on a multicore node enhanced with multiple GPU accelerators. In *25th IEEE International Symposium on Parallel and Distributed Processing*, IPDPS'11, pages 932 –943, May 2011.

[3] E. Agullo, C. Augonnet, J. Dongarra, H. Ltaief, R. Namyst, S. Thibault, and S. Tomov. Faster, Cheaper, Better – a Hybridization Methodology to Develop Linear Algebra Software for GPUs. In W. mei W. Hwu, editor, *GPU Computing Gems*, volume 2. Morgan Kaufmann, Sep. 2010.

[4] AMD. The industry-changing impact of accelerated computing. AMD Whitepaper, Advance Micro Devices, 2008. 09/04/2011.

[5] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov, and D. Sorensen. *LAPACK's user's guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1992.

[6] 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*, 23(2):187–198, 2011.

[7] M. Baboulin, S. Donfack, J. Dongarra, L. Grigori, A. Rémy, and S. Tomov. A class of communication-avoiding algorithms for solving general dense linear systems on CPU/GPU parallel machines. *Procedia CS*, 9:17–26, 2012.

[8] G. Barlas, A. Hassan, and Y. A. Jundi. An analytical approach to the design of parallel block cipher encryption/decryption: A CPU/GPU case study. In *Proceedings of the 19th International Euromicro Conference on Parallel, Distributed and Network-Based Processing*, PDP '11, pages 247–251, Washington, DC, USA, 2011. IEEE Computer Society.

[9] A. Baumann, P. Barham, P.-E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach, and A. Singhania. The multikernel: a new OS architecture for scalable multicore systems. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, SOSP '09, pages 29–44, New York, NY, USA, 2009. ACM.

[10] L. S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley. *ScaLAPACK user's guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1997.

[11] B. C. Budge, J. C. Anderson, C. Garth, and K. I. Joy. A hybrid CPU-GPU implementation for interactive ray-tracing of dynamic scenes. Technical Report CSE-2008-9, University of California, Davis Computer Science, 2008.

[12] S. Chakrabarti, J. Demmel, and K. Yelick. Modeling the benefits of mixed data and task parallelism. In *Proceedings of the Seventh Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '95, pages 74–83, New York, NY, USA, 1995. ACM.

[13] R. Dorrigiv, A. López-Ortiz, and A. Salinger. Optimal speedup on a low-degree multi-core parallel architecture (LoPRAM). In *Proceedings of the Twentieth Annual Symposium on Parallelism in Algorithms and Architectures*, SPAA '08, pages 185–187, New York, NY, USA, 2008. ACM.

[14] P. Ezzatti, E. Quintana-Ortí and, and A. Remon. High performance matrix inversion on a multi-core platform with several GPUs. In *Parallel, Distributed and Network-Based Processing (PDP), 2011 19th Euromicro International Conference on*, pages 87 –93, Feb. 2011.

[15] J. R. Humphrey, D. K. Price, K. E. Spagnoli, A. L. Paolini, and E. J. Kelmelis. Cula: hybrid GPU accelerated linear algebra routines. In *Proc. of SPIE Defense and Security Symposium (DSS)*, April 2010.

[16] Khronos OpenCL Working Group. *The OpenCL Specification, version 1.2.19*, 14 November 2012.

[17] J. Kurzak, R. Nath, P. Du, and J. Dongarra. An implementation of the tile QR factorization for a GPU and multiple CPUs. In *Proceedings of the 10th international conference on Applied Parallel and Scientific Computing - Volume 2*, PARA'10, pages 248–257, Berlin, Heidelberg, 2012. Springer-Verlag.

[18] A. López-Ortiz, A. Salinger, and R. Suderman. Toward a generic hybrid CPU-GPU parallelization of divide-and-conquer algorithms. In *Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2013 IEEE 27th International*, pages 601–610, 2013.

[19] M. D. McCool, K. Wadleigh, B. Henderson, and H.-Y. Lin. Performance evaluation of GPUs using the RapidMind development platform. In *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, SC '06, New York, NY, USA, 2006. ACM.

[20] J. Nickolls, I. Buck, M. Garland, and K. Skadron. Scalable parallel programming with CUDA. *Queue*, 6(2):40–53, Mar 2008.

[21] E. B. Nightingale, O. Hodson, R. McIlroy, C. Hawblitzel, and G. Hunt. Helios: heterogeneous multiprocessing with satellite kernels. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, SOSP '09, pages 221–234, New York, NY, USA, 2009. ACM.

[22] NVIDIA. Dynamic parallelism in CUDA, 2012. Retrieved on 01/19/2013.

[23] M. Papadrakakis, G. Stavroulakis, and A. Karatarakis. A new era in scientific computing: Domain decomposition methods in hybrid CPU-GPU architectures. *Computer Methods in Applied Mechanics and Engineering*, 200(13-16):1490–1508, Mar. 2011.

[24] G. Quintana-Ortí, F. D. Igual, E. S. Quintana-Ortí, and R. A. van de Geijn. Solving dense linear systems on platforms with multiple hardware accelerators. In *Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '09, pages 121–130, New York, NY, USA, 2009. ACM.

[25] A. L. Rosenberg and R. C. Chiang. Toward understanding heterogeneity in computing. In *24th IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2010, Atlanta, Georgia, USA*, pages 1–10, 2010.

[26] L. Seiler, D. Carmean, E. Sprangle, T. Forsyth, P. Dubey, S. Junkins, A. Lake, R. Cavin, R. Espasa, E. Grochowski, T. Juan, M. Abrash, J. Sugerman, and P. Hanrahan. Larrabee: A many-core x86 architecture for visual computing. *IEEE Micro*, 29(1):10–21, 2009.

[27] J. E. Stone, D. Gohara, and G. Shi. OpenCL: A parallel programming standard for heterogeneous computing systems. *Computing in Science and Engineering*, 12:66–73, 2010.

[28] L. Surhone, M. Tennoe, and S. Henssonow. *Intel Array Building Blocks*. VDM Verlag Dr. Mueller AG & Co. Kg, 2010.

[29] G. Teodoro, R. Sachetto, O. Sertel, M. Gurcan, W. Meira, U. Catalyurek, and R. Ferreira. Coordinating the use of GPU and CPU for improving performance of compute intensive applications. In *Cluster Computing and Workshops, 2009. CLUSTER '09. IEEE International Conference on*, pages 1 –10, Aug.-Sep. 2009.

[30] S. Tomov, J. Dongarra, and M. Baboulin. Towards dense linear algebra for hybrid GPU accelerated manycore systems. *Parallel Comput.*, 36:232–240, June 2010.

[31] S. Tomov, J. Dongarra, P. Du, and R. Nath. Magma version 0.2 user guide. MAGMA, 2009. 09/06/2011.

[32] S. Tomov, R. Nath, and J. Dongarra. Accelerating the reduction to upper hessenberg, tridiagonal, and bidiagonal forms through hybrid GPU-based computing. *Parallel Comput.*, 36(12):645–654, Dec. 2010.

[33] S. Venkatasubramanian and R. W. Vuduc. Tuned and wildly asynchronous stencil kernels for hybrid CPU/GPU systems. In *Proceedings of the 23rd international conference on Supercomputing*, ICS '09, pages 244–255, New York, NY, USA, 2009. ACM.

[34] C. Vömel, S. Tomov, and J. Dongarra. Divide and conquer on hybrid GPU-accelerated multicore systems. *SIAM Journal on Scientific Computing*, 34(2):C70–C82, 2012.