Pattern-guided Big Data Processing on Hybrid Parallel Architectures

Fahad Khalid, Frank Feinbube, Andreas Polze

Operating Systems and Middleware Group
Hasso Plattner Institute for Software Systems Engineering
Prof.-Dr.-Helmert-Str. 2-3
14482 Potsdam
fahad.khalid@hpi.uni-potsdam.de
frank.feinbube@hpi.uni-potsdam.de
andreas.polze@hpi.uni-potsdam.de

Abstract: The advent of hybrid CPU-GPU architectures has significantly increased the number of raw FLOP/s. However, it is not obvious how these can be put to use when processing Big Data. In this paper, we present an approach for designing Big Data simulations for hybrid architectures, which is based on a hierarchal application of design patterns in parallel programming. We provide a detailed account of the step by step approach that results in efficient utilization of processing and memory resources, while simultaneously improving developer productivity. Finally, we present our vision of automated tools that will further simplify the development of efficient parallel implementations for Big Data processing on hybrid architectures.

1 Introduction

Recent advances in experimental and and computational natural sciences have led to an unprecedented increase in data processing requirements. In domains such as biology, new insights into processes such as regulation of gene transcription, metabolism, signal transduction, and protein-protein interaction, have led to new models that require more complex computations on larger datasets for realistic simulations. Similarly, in geophysics, the increasing amounts of seismic data being captured due to a large number of sensors placed in seismically active regions; and a multitude of innovations in other fields have resulted in the generation of very large datasets that are used by simulations in order to produce accurate predictions. Such simulations not only require powerful processing resources, but demand novel and effective ways of managing very large amounts of data.

Algorithms used in such simulations can be broadly categorized based on whether they exhibit regular or irregular data access patterns. This paper focuses solely on algorithms with regular data access patterns that result in decomposable input domains. The current generation of hardware architectures provides performance high enough to satisfy compute requirements for such simulations. However, processing very large data sets does not just require compute performance. It requires that the simulations be able to efficiently manage large amounts of data subject to the limited main memory and cache sizes. This situation is further complicated in hybrid architectures where the system consists of both CPUs and accelerators such as GPUs, whose massively parallel architecture is particularly suited to data parallel applications. In such cases, each processor type has an associated memory hierarchy different from the other. Even though such hybrid architectures contribute a high number of FLOP/s, managing high data volumes becomes quite challenging.

In this paper we propose an approach to effectively manage the processing of very large data sets on shared-memory hybrid architectures with CPUs and GPUs. We argue that a hierarchical and methodical application of *patterns for parallel programming* [KMMS10] can be used to maximize system performance, and provide a means to efficiently manage very large data sets. Moreover, we present an approach – complimented by our vision – that helps improve developer productivity; thereby reducing the overall cost of development for hybrid architectures.

The paper is organized as follows: Section 2 describes the pattern-based design approach for efficient resource utilization in hybrid architectures. Each subsection elaborates on a different level of hierarchy and how it can be applied. This is followed by a discussion on the applicability and limitations of the approach in Section 3. Section 4 describes the novel *Architecture-based Algorithm Decomposition Approach* proposed in this paper. This is followed by a discussion on how tools can assist in increasing developer productivity by automatically discerning parallel design patterns and exposing potential parallelism in serial code.

Note: Throughout the rest of the document, we use the term *Host* to refer to a combination of one or more CPUs available in a hybrid system. We use the term *Device* to refer to the GPU available in the same system.

2 Patterns, Productivity and Efficient Hybrid-Resource Utilization

In this section we describe a design approach based on hierarchical application of *patterns for parallel programming* to data parallel applications with regular data access patterns. We hypothesize that not only can our approach be used to improve simulation performance on a hybrid architecture, it also provides the foundation for developing frameworks and automation tools that can improve developer productivity. We call this method the *Efficient Hybrid-Resource Utilization (EHRU)* approach. This approach, as described in this paper, is based on previous work [KNTP13, KFP14].

2.1 Simulation as Pipeline

Let us consider a typical simulation scenario consisting of the following three steps:

1. Input data is read from file

2. Input data is processed using one or more computational kernels
3. Output is written to file

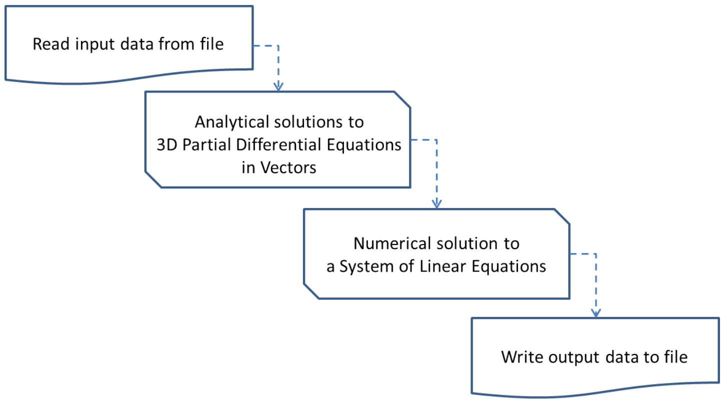
Read input data from file
Analytical solutions to 3D Partial Differential Equations in Vectors
Numerical solution to a System of Linear Equations
Write output data to file

Figure 1: A hypothetical simulation depicted as pipeline.

A pictorial representation of the above mentioned steps is provided in Figure 1. In addition to the two file I/O stages, the simulation presented in Figure 1 comprises of two different computational kernels; each constituting a pipeline stage. Let us assume that the input dataset is very large, and only a fraction of the dataset can fit into main memory. In this case, the dataset can be partitioned into several small chunks, where each chunk can easily fit into main memory. Then, the first stage of the pipeline reads one chunk at a time and passes it on to the next stage. Each stage processes output from the previous stage and passes its output to the next stage, until the final result is written to file. This mechanism makes it possible to process a dataset too large to fit into the main memory.

It is worthwhile mentioning that a single chunk that completely fills the memory essentially serializes the pipeline. Therefore, determining the appropriate size of the input data chunk requires analyzing various properties of the different stages of the pipeline. We refer the reader to [KFP14] for a detailed account of optimizing pipeline performance.

The concept of parallel pipelining [DFF$^+$02] is well established in parallel computing. Our intention here is not introduce pipelining as a contribution, but rather to lay the foundation for the use of the *pipeline* pattern [KMMS10, MRR12] for efficient big data processing on hybrid architectures.

2.2 Data Partitioning and Processing on a Hybrid Architecture

Representing the entire simulation as a pipeline is considered as the top-level pattern in our proposed hierarchy of patterns. The file I/O stages remain at this level of hierarchy.

The processing stages, however, can be further decomposed into two parts, i.e., 1) *Host-only processing*, and 2) *Hybrid Pipeline*. In order to distribute the input data chunk over the *Host-only processing* implementation and the *Hybrid Pipeline* implementation, the *geometric decomposition* [KMMS10] pattern and/or the *partition* [MRR12] pattern can be used. Once the data is partitioned, *Host-only processing* and *Hybrid pipeline* work as two parallel forks. The number of *Host* threads assigned to each fork depends on the nature of the algorithm and the resulting performance considerations. Once both forks have finished processing the assigned chunks of data, a join followed by a reduction can be used to consolidate the separately computed results. Therefore, at this level, in addition to geometric decomposition and partition, *fork/join* [MRR12] and *reduction* [MRR12] patterns are utilized. This constitutes level 2 of the pattern hierarchy.

```
(L2) EHRU { Patterns: Geometric decomposition, Pipeline, Fork/Join, Reduction
    (L3) Hybrid Pipeline { Patterns: Pipeline
            (L3.1) Device { Patterns: Partition, <Stencil, Map>
                                  ...
                    }
            (L3.2) Post-processing { Patterns: <Scan, Reduction>
                                  ...
                    }
         }
    (L3) Host-only { Patterns: Geometric decomposition, <<Stencil>>, <<Scan>>
                          ...
          }
}
```

Figure 2: Pattern hierarchy.

2.3 Host-only Processing

The total number of available *Host* threads are divided into two groups; one for *Host-only* processing and the other for *Hybrid pipeline*. *Host-only* processing is based on an implementation that combines all the processing stages of the outer pipeline into a single *Host*-only execution sequence. This implementation can then use the *loop parallelism* [KMMS10] pattern and/or the *fork/join* pattern to process its share of data in parallel. High-level parallel programming models such as OpenMP [Boa11], Intel TBB [Rei07], Cilk [Rob13], OpenCL [Gro11] etc., can be used by the developer to easily implement these patterns. *Host-only processing* and *Hybrid pipeline* constitute level 3 of the pattern hierarchy.

2.4 Hybrid Pipeline

We use the term *Hybrid Pipeline* for an implementation of the parallel pipeline pattern where different stages of the pipeline are executed on different processor types, e.g., some on the *Host* and others on the *Device*. For an outer pipeline similar to the one depicted in Figure 1, where the processing algorithm comprises two stages, the two stages can be characterized as the *producer-consumer* dynamic. Such a produce-consumer dynamic can be implemented using a 3-stage hybrid pipeline, as shown in Figure 3.

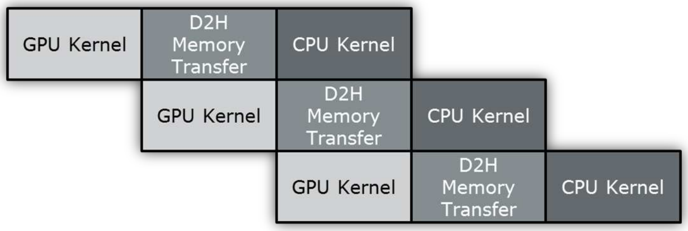

Figure 3: A 3-stage hybrid parallel pipeline.

Let us assume that the first processing stage is the most compute intensive stage of the two, and therefore constitutes the part of the simulation where most of the processing time is spent. Then, in order to improve the performance of this stage, the massively parallel nature of the *Device* can be exploited by implementing this stage as a *Device* kernel.

Results from the *Device* kernel can then be fed into the next stage which comprises of a less compute intensive post-processing algorithm. Since the computational requirements for this stage are not as high, sufficient performance can be gained by using a *Host*-only parallel implementation. Here, we assume that the time required to process this stage as a *Host*-only parallel implementation is less than the time required to process the *Device* kernel stage.

Since the two above mentioned stages execute on different processor types, results from the first stage must be copied from the *Device* memory to the *Host* memory. Such transfer of memory can be time consuming, and therefore must be overlapped with computation. Consequently, it is important to introduce a third stage between the two stages which implements the asynchronous transfer of results from *Device* memory to the *Host* memory.

All the above mentioned stages, when combined, form a 3-stage hybrid pipeline. The advantages of such a hybrid pipeline are two-fold. Assuming that the post-processing stage consumes less time than the *Device* stage, the time taken by the post-processing stage is outweighed by the *Device* stage for most of the pipeline iterations. For a large enough dataset, the post-processing stage works at almost no cost in terms of time. Moreover, since the post-processing stage is implemented on the *Host* using high level programming models, the development effort is much lower. Therefore, the hybrid pipeline provides efficient processing and high productivity. A generic framework for hybrid pipelining was

Table 1: Lists of suitable and unsuitable dwarfs.

Suitable Dwarfs	Unsuitable Dwarfs
Dense Linear Algebra	Sparse Linear Algebra
Structured Grids	Unstructured Grids
Monte Carlo	Graph Traversal

presented in [KFP14].

3 Feasibility and Limitations of the EHRU Approach

As mentioned in Section 1, our approach primarily focuses on data parallel applications with regular access patterns. In this section, we show that there are several broad categories of algorithms that fit these criteria. Moreover, we also highlight categories of algorithms for which our approach is not suitable. Both these categories are based on Berkeley Dwarfs [ABC+06]. Table 1 provides a comparative listing of dwarfs for which *EHRU* approach is suitable and unsuitable.

3.1 Applications for which the EHRU Approach is Suitable

The most important parameter that contributes to the applicability of the *EHRU* approach is decomposability of data. If the input data can be easily decomposed into several chunks that can be processed using synchronization-free parallelism, the *EHRU* approach can be used, regardless of the access patterns within the chunks.

Linear Algebra operations on dense vectors and matrices constitute one category of algorithms for which the *EHRU* approach is suitable. E.g., two very large dense matrices can be multiplied block-wise [Str03], without requiring communication between different processes that operate on different blocks. In certain simulations, Finite Element [ZM71] or Boundary Element [BB81] methods are used, where the same computations are applied to most elements in a finite grid. A structured grid can often be easily divided into many different parts that can be processed in parallel without the need for synchronization. Final results in such applications are computed by applying a post-processing reduction operation.

3.2 Applications for which the EHRU Approach is not Suitable

Many Big Data applications operate on input datasets which cannot be easily partitioned into chunks that can be processed independently. One category of such problems are those that operate on graph structures. Graph partitioning in itself is a hard problem [BMS+13].

Therefore, for simulations involving graph structures, it is not often possible to partition the input datasets in a way that chunks can be processed in a pipeline. This is because an element in one chunk might require access to elements spread over several other chunks.

Another possible negative case is where the simulation generates huge amounts of data at runtime, and data generated at each iteration is used as input to the next iteration. Therefore, the memory requirements increase at runtime. One example from the domain of computational geometry is the problem of *extreme ray enumeration in a polyhedral cone* [MRTT53]. In this problem, the memory consumption grows exponentially over each iteration. Moreover, the output generated during each iteration cannot be easily partitioned, i.e., every process requires access to the entire data. Therefore, pipelining the input data is not a feasible solution.

4 Architecture-based Algorithm Decomposition

The hybrid pipeline approach outlined in Section 2.4 raises an important question, *"how does one decide which stage of the hybrid pipeline to implement on which architecture?"*. In this section, we argue that it should be possible to structure the pipeline flow in such a way that different stages are executed by either the *Device* or the *Host* depending on which architecture is more suitable for that part of the kernel. In the following subsections we highlight the characteristics of a computational kernel that make it suitable for execution on a certain architecture. We then propose, that based on these characteristics, an algorithm can be decomposed into multiple kernels, where each kernel is suitable for execution on one specific processor type in a hybrid architecture. The overall approach is depicted in Figure 4.

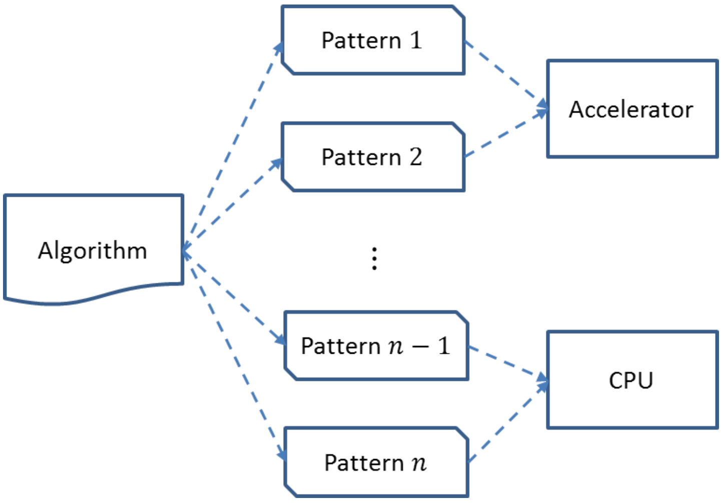

Figure 4: Architecture-based algorithm decomposition.

4.1 Characteristics of Computational Kernels

Host architectures (i.e., CPUs such as Intel Core i7, Intel Xeon series etc.) are equipped with a feature rich Instruction Set Architecture (ISA), Vector processing units, and advanced features such as branch prediction. This makes it possible for the *Host* to efficiently perform all kinds of computations. The major disadvantage as compared to *Device* architectures is the lack of massive parallelism.

Device architectures have the following major limitations:

- A high *Degree of Parallelism (DoP)* in a kernel is a fundamental requirement. If the degree of parallelism is not high enough, compute cycles are not fully utilized.
- Caches are much smaller on the *Device*. Therefore, kernels with low *arithmetic intensity* [PH13] are not particularly suitable.
- *Device* architectures do not support branch prediction, and are bounded by a minimum number of threads that must execute the same instruction in a single clock cycle. Therefore, excessive *control divergence* in the kernel can lead to wasted cycles.

It is argued in the following sub-sections that *Device* kernels for which the processing is dominated by one or more of the above mentioned limitations, can gain performance improvement if the appropriate parts of the kernel are executed on the suitable architecture.

4.2 Algorithm Decomposition and Design Patterns

An algorithm can be decomposed based on architectural features if the following conditions are satisfied:

- At least two design patterns can be identified, where one pattern is suitable for the *Device* architecture, and the other one is suitable for the *Host* architecture. Each pattern can then be implemented as a stage of computation.
- The *pipeline* pattern is applicable across the two stages of computation.

The design patterns for parallel programming can be used to identify the data and control flow in a computational kernel. This makes it possible to see whether a certain kernel would be suitable for a given architecture. The following subsections describe the patterns that are suitable for *Device* architectures, as well as those that are not suitable for *Device* architectures.

4.2.1 Patterns Suitable for *Device* Architectures

Some of the important patterns suitable for execution on the *Device* are:

- **Map**: The Map [MRR12] pattern can be seen as a fully unrolled loop, where each iteration of the loop is mapped to a different processing unit of a massively parallel processor. There are no dependencies between the loop iterations, which implies that each processing unit reads input values and writes output values that are completely independent of all other processing units. The degree of parallelism is proportional to the number of loop iterations, which means that the hardware utilization is maximal for a large number of iterations.
- **Stencil**: Similar to Map, there are no output related dependencies amongst the processing units. This pattern is also similar to a loop without dependencies. In the Stencil [MRR12] pattern, however, a processing unit needs access to input values from the neighboring processing units as well. In fact, Map can be seen as a special case of Stencil with the restriction that each processing unit only reads its own corresponding input values.

Both these patterns map well onto the massively parallel *Device* architecture. This is because each processing unit can perform computations completely independent of all other units. There is no need for elaborate locking and synchronization primitives, which could slow down the computation.

4.2.2 Patterns Suitable for *Host* Architectures

As mentioned in Section 4.1, the *Host* architecture is suitable for all computational kernels. The only limitation is that massive parallelism is not available on these architectures. The following patterns are recommended for execution on the *Host* because they do not exploit the massively parallel architecture of the *Device* efficiently.

- **Reduce**: The Reduce [MRR12] pattern is employed when a large number of values have to be reduced to a single value. E.g., a vector of 10 values is reduced to a scalar by summing all the elements of the vector. Furthermore, the Reduce pattern can be applied to any data type on which an associative binary operator is defined. The most efficient method for parallel reduction follows computation in a tree structure. For a sum reduction, it starts with all the values at the leaves, and proceeds upwards to the root by applying the operator at each level, and halving the number of operands. Eventually, only the final scalar value is left at the root, which is the desired result.
- **Scan**: An example of the Scan [MRR12] pattern is the prefix-sum operation. In this case, the output vector is the same size as the input vector. Therefore, the tree structure consists of two sweeps: 1) *Up-sweep* from the leaves to the root to compute partial reductions, and 2) *Down-sweep* from the root back to the leaves to complete the scan.

Both the Reduce and Scan patterns share the same problem when it comes to efficient resource utilization. The degree of parallelism varies at each step of computation. In Reduce, e.g., the number of processing units that can be used at the first step is equal to the number of elements in the vector. For a large vector, this amounts to maximum resource utilization. However, at each successive step, the number of processing units required is halved; eventually leading to the point where most of the compute resources are idle.

5 Tool-guided Parallelization for Hybrid Architectures

As described in Section 3, the overall approach presented in this paper is suited primarily to problems with easily decomposable data and regular access patterns. It follows from these assumptions, that for such applications, most loops are composed of affine iteration spaces [LL97], as well as affine mapping functions [LL97] from iteration index to array element. This makes it possible to apply the existing theory of dependence analysis and loop transformation to these problems.

5.1 Discerning Parallel Patterns from Serial Source Code

The *architecture-based algorithm decomposition* approach (described in Section 4) relies on the assumption that the developer can easily discern parallel design patterns from serial source code. This, however, requires a significant amount of knowledge and experience on part of the developer. Therefore, we argue that an automated tool that can discern patterns from serial source code would further improve developer productivity.

It is our conjecture that the fundamental mathematical property that distinguishes different patterns is the *type of dependence among the iterations* of the corresponding *for* loop. The type of dependence can be computed using the already established theory of dependence analysis [KA01]. The result can then be mapped onto the corresponding pattern. A tool capable of discerning parallel patterns from serial source code will not only improve developer productivity, it will also connect theoretical/compiler level concepts to higher level abstractions of patterns that are traditionally studied at the level of software engineering.

5.2 Suggestions for Parallelization for Hybrid Architectures

Given our earlier assumption regarding affine spaces and functions, we propose that instead of the developer having to discover all different levels of parallelism, a tool imbued with the knowledge of patterns as well as discerning parallelism, can assist the developer by highlighting possibly parallelizable code segments, as well as suggesting appropriate optimizing transformations. Intel Parallel Advisor [BCS12] is a commercially available tool with similar features. However, neither does Intel Parallel Advisor discern patterns from source code, nor is it applicable to hybrid architectures with GPUs. We are currently working towards the development of such a tool for hybrid architectures.

6 Related Work and Our Contribution

As determined in a thorough literature review [KFP14] on the subject, apart from overlapping of computation and transfer of data from GPU memory to CPU memory, pipelining has also been used in hybrid architectures for improving resource utilization and load balancing. Most pipeline applications for hybrid architectures are focused on solving a particular problem. E.g., much work has been done in utilizing pipelining for hybrid *MapReduce* implementations [CQD$^+$13, SO11]. In [CHA12], an efficient *MapReduce* scheduling approach is presented for a coupled CPU-GPU chip. This approach makes it possible to dynamically divide Map tasks onto CPU and GPU, thereby utilizing both types of processing resources. The *Moim* [XKB] framework for *MapReduce* supports simultaneous execution of the reduce phase on both CPU and GPU. Other than *MapReduce*, a 3-stage hybrid CPU-GPU pipeline has also been used for solving eigenvector and eigenvalue problems [GGV12].

As mentioned in Section 5.2, Intel Parallel Advisor [BCS12] is a commercially available tool that guides a developer through the process of parallelizing existing serial source code. There have also been other efforts with similar goals. Prospector [KKL10] is a tool designed to assist programmers in parallelizing legacy serial applications developed by other programmers. For GPUs, GROPHECY [MMK$^+$11] is a performance projection framework that uses skeletonized pieces of CPU source code to determine if the performance gain from porting the application to GPU is worth the development effort.

To the best of our knowledge, the *Architecture-based Algorithm Decomposition* is a novel idea introduced in this paper. Moreover, despite various commendable efforts in developing automated tools for assisting in code parallelization for either CPUs or GPUs, one has yet to be developed for hybrid architectures. We believe that our ideas presented in Section 5 will help us develop an effective tool for parallelization assistance on hybrid architectures. Our hierarchical application of parallel patterns for efficient processing of very large datasets is also original, even though the general theme of optimizing resource utilization for hybrid architectures has already been explored from other perspectives.

7 Summary and Future Work

We have presented our *Efficient Hybrid-Resource Utilization* approach for processing very large datasets on hybrid CPU-GPU based architectures. It comprises data decomposition and distribution amongst different processor types within a shared-memory system. System efficiency and developer productivity are both improved by employing hierarchical application of patterns for parallel programming. While this approach has partly been implemented and published elsewhere, in this paper, we have extended this idea as a foundation for *Architecture-based Algorithm Decomposition*. We have argued that the suitability of execution of a particular computational kernel on a particular processor type can be determined by inspecting the pattern(s) of which the kernel consists. Certain characteristics of a pattern make it suitable for execution on one or the other processor type in a hybrid architecture. In certain cases, in addition to improving system performance, this information can save the development effort required for implementing *Device* kernels, thereby improving developer productivity.

We have further argued, that it is plausible to assume, that for data parallel applications with regular access patterns, it is possible to use dependence analysis for discerning parallel patterns from serial source code. Automation of this process can further assist in *Architecture-based Algorithm Decomposition*. Moreover, it is plausible to assume that for such applications, an automated tool can be developed that highlights potential parallelism in a serial application, and guides the developer through the process of parallelization for hybrid architectures.

We are currently in the process of developing theoretical foundations for *discerning parallel patterns from serial source code*. Finishing this project is our short-term objective. The tool resulting from this project will then be applied to *Architecture-based Algorithm Decomposition*. Eventually, we hope to have an automated tool for assisting the developer in developing parallel applications for hybrid parallel architectures.

References

[ABC$^+$06] Krste Asanovic, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David A. Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams, and Katherine A. Yelick. The Landscape of Parallel Computing Research: A View from Berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, Dec 2006.

[BB81] Prasanta Kumar Banerjee and Roy Butterfield. *Boundary element methods in engineering science*, volume 17. McGraw-Hill London, 1981.

[BCS12] Stephen Blair-Chappell and Andrew Stokes. *Parallel Programming with Intel Parallel Studio XE*. John Wiley & Sons, 2012.

[BMS$^+$13] Aydin Buluç, Henning Meyerhenke, Ilya Safro, Peter Sanders, and Christian Schulz. Recent Advances in Graph Partitioning. *CoRR*, abs/1311.3144, 2013.

[Boa11] OpenMP Architecture Review Board. OpenMP Application Program Interface. Standard specification, July 2011.

[CHA12] Linchuan Chen, Xin Huo, and Gagan Agrawal. Accelerating mapreduce on a coupled cpu-gpu architecture. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, page 25. IEEE Computer Society Press, 2012.

[CQD$^+$13] Yi Chen, Zhi Qiao, Spencer Davis, Hai Jiang, and Kuan-Ching Li. Pipelined Multi-GPU MapReduce for Big-Data Processing. In *Computer and Information Science*, pages 231–246. Springer, 2013.

[DFF$^+$02] Jack Dongarra, Ian Foster, Geoffrey C. Fox, William Gropp, Ken Kennedy, Linda Torczon, and Andy White, editors. *The Sourcebook of Parallel Computing*. Morgan Kaufmann, 2002.

[GGV12] Michael T Garba and HORACIO GONZÁLEZ-VÉLEZ. Asymptotic peak utilisation in heterogeneous parallel CPU/GPU pipelines: a decentralised queue monitoring strategy. *Parallel Processing Letters*, 22(02), 2012.

[Gro11] Khronos OpenCL Working Group. The OpenCL Specification. Standard specification, December 2011.

[KA01] Ken Kennedy and John R Allen. Optimizing compilers for modern architectures: a dependence-based approach. 2001.

[KFP14] Fahad Khalid, Frank Feinbube, and Andreas Polze. Hybrid CPU-GPU Pipeline Framework. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA)*, 2014. To appear.

[KKL10] Minjang Kim, Hyesoon Kim, and Chi-Keung Luk. Prospector: A dynamic data-dependence profiler to help parallel programming. In *HotParâ10: Proceedings of the USENIX workshop on Hot Topics in parallelism*, 2010.

[KMMS10] Kurt Keutzer, Berna L. Massingill, Timothy G. Mattson, and Beverly A. Sanders. A design pattern language for engineering (parallel) software: merging the PLPP and OPL projects, 2010.

[KNTP13] Fahad Khalid, Zoran Nikoloski, Peter Tröger, and Andreas Polze. Heterogeneous Combinatorial Candidate Generation. In Felix Wolf, Bernd Mohr, and Dieter Mey, editors, *Euro-Par 2013 Parallel Processing*, volume 8097 of *Lecture Notes in Computer Science*, pages 751–762. Springer Berlin Heidelberg, 2013.

[LL97] Amy W. Lim and Monica S. Lam. Maximizing Parallelism and Minimizing Synchronization with Affine Transforms. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '97, pages 201–214, New York, NY, USA, 1997. ACM.

[MMK$^+$11] Jiayuan Meng, Vitali A Morozov, Kalyan Kumaran, Venkatram Vishwanath, and Thomas D Uram. GROPHECY: GPU performance projection from CPU code skeletons. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, page 14. ACM, 2011.

[MRR12] Michael McCool, James Reinders, and Arch Robison. *Structured parallel programming: patterns for efficient computation*. Elsevier, 2012.

[MRTT53] T. S. Motzkin, H. Raiffa, G. L. Thompson, and R. M. Thrall. The double description method. *Contributions to the Theory of Games*, 2:51–73, 1953.

[PH13] David A Patterson and John L Hennessy. *Computer organization and design: the hardware/software interface*. Newnes, 2013.

[Rei07] James Reinders. *Intel threading building blocks: outfitting C++ for multi-core processor parallelism*. O'Reilly Media, Inc., 2007.

[Rob13] Arch D Robison. Composable Parallel Patterns with Intel Cilk Plus. *Computing in Science & Engineering*, 15(2):0066–71, 2013.

[SO11] Jeff A Stuart and John D Owens. Multi-GPU MapReduce on GPU clusters. In *Parallel & Distributed Processing Symposium (IPDPS), 2011 IEEE International*, pages 1068–1079. IEEE, 2011.

[Str03] Gilbert Strang. *Introduction to linear algebra*. SIAM, 2003.

[XKB] Mengjun Xie, Kyoung-Don Kang, and Can Basaran. Moim: A Multi-GPU MapReduce Framework.

[ZM71] Olgierd Cecil Zienkiewicz and PB Morice. *The finite element method in engineering science*, volume 1977. McGraw-hill London, 1971.