Exposing Hidden Performance Opportunities in High Performance GPU Applications

Benjamin Welton and Barton P. Miller
Computer Sciences Department
University of Wisconsin - Madison
Madison, WI 53706
{welton,bart}@cs.wisc.edu

Abstract—**Leadership class systems with nodes containing many-core accelerators, such as GPUs, have the potential to increase the performance of applications. Effectively exploiting the parallelism provided by many-core accelerators requires developers to identify where accelerator parallelization would provide benefit and ensuring efficient interaction between the CPU and accelerator. In the abstract, these issues appear straightforward and well understood. However, we have found that significant untapped performance opportunities exist in these areas even in well-known, heavily optimized, real world applications created by experienced GPU developers. These untapped performance opportunities exist because accelerated libraries can create unexpected synchronization delay and memory transfer requests, interaction between accelerated libraries can cause unexpected inefficiencies when combined, and vectorization opportunities can be hidden by the structure of the program. In applications we have studied (Qball, QBox, Hoomd-blue, LAMMPs, and cuIBM), exploiting these opportunities resulted in reduction of their execution time by 18%-87%. In this work, we provide concrete evidence of the existence and impact that these performance issues have on real world applications today. We characterize the missed performance opportunities we have identified by their underlying cause and describe a preliminary design of detection methods that can be used by performance tools to identify these missed opportunities.**

Index Terms—**Distributed Computing, Many-Core Processing, Graphics Processing Unit, Hidden Performance Opportunities, Performance Tools**

I. Introduction

As many-core accelerators have become standard on high performance computing platforms, developers have had to adapt their applications to exploit the additional parallelism afforded by many-core architectures. The adaptation process begins with the identification of code suitable for parallelization, the writing of an efficient many-core parallelizations of that code, handling the interaction between the CPU and many-core device, and ends with the integration of the new many-core component into existing CPU code. Developers often further optimize their applications by iterating over this process many times.

When viewed in isolation, each stage of the adaptation process appears straight forward and well understood. However, when adapting and tuning real-world applications, the identification of performance opportunities within each stage becomes increasingly difficult. The sheer size and complexity of real-world code bases, which can number in the hundreds of thousands of lines of code or more, makes manual identification of performance opportunities increasingly difficult. For instance, a highly vectorizable sequential memory access pattern can be hidden from source code analysis when it exists because of memory layout changes in another part of the program. The code bases of real-world applications are also frequently changing, the result of these changes can make once efficient interactions identified by previous optimization passes of the application inefficient. The reliance of developers on many-core accelerated libraries to add functionally to their applications increases the difficulty of optimization further. These libraries are efficient internally but when combined with other many-core accelerated codes create an undesirable interaction patterns between the CPU and the accelerator. Real-world applications also typically support many different many-core accelerator types (such as OpenMP, CUDA, and Phi). Assumptions made by application developers on usage of these accelerator types can lead to bad behavior when they are not true (such as assuming that multiple modes of acceleration will not be used together). Finally, due to a lack of performance tools that provide visibility into the interactions between CPU and accelerator across multiple components, developers are on their own to evaluate their application to find any opportunities that may exist.

The end result of the increased complexity makes seeming straight forward performance opportunities difficult to identify in practice by skilled developers. The untapped performance opportunities we have identified in a collection of highly optimized real-world applications take the form of missed unobvious many-core parallelization opportunities and inefficiencies in handling interactions with the accelerator, such as duplicate memory copies and unnecessary synchronization. These issues were identified with a combination of source code review and manual instrumentation to gain details about the runtime of functions within the applications and memory structure; manual corrections were inserted when an issue was identified. In our initial experiments with the real-world GPU applications run on Oak Ridge National Laboratorys Cray Titan supercomputer (Table I), we have found that the exploitation of these performance opportunities reduces application execution time between 19-85%.

We needed to use a manual process to identify these problems because current performance tools do not provide

Application Name (Version)	Organization	Application Description	Original Runtime (Min:Sec)	Runtime Reduction	Problems Found
Hoomd-Blue [3] (v1.1.1)	Univ of Michigan	Molecular Dynamics Particle Simulator	08:36	37%	SYN
Qbox [16] (v1.63.5)	UC Davis	First Principal of Molecular Dynamics	38:54	85%	DD, SYN
QBall [11] (Apr 24 2017)	LLNL	First Principal of Molecular Dynamics (enhanced version of qbox)	67:55	87%	DD, SYN
LAMMPs [34] (Mar 31 2017)	Sandia	Molecular Dynamics Particle Simulator	03:34	18%	MP
cuIBM [21] (Sep 21 2016)	GWU	Computational Fluid Dynamics	31:42	27%	SYN, JT

MP: Unobvious missed parallelization, DD: Duplicate Data Transfers, SYN: Synchronization, JT: Just-In-Time GPU Compilation

TABLE I: Applications improved by adding parallelism and correcting inefficient behavior

information that is precise enough to detect the presence of these problems nor accurate enough to explain the cause. Some missing features of existing profilers and tracers include not examining the contents of memory transfers, not associating synchronization operations with the data that they protect, and not examining the actual memory-access pattern of a loop at run time to determine its vectorizability. As a result, a tool such as nvprof [30] can identify in some cases time consuming data transfers, synchronizations, and loops but does not provide information such as whether these operations are necessary or can be improved.

The goal of our work is to help developers reveal and ultimately correct these inefficiencies in their applications. Our contributions include: (1) characterizing the missed performance opportunities in many-core applications and why they are difficult to identify, (2) show the performance benefit of correcting these issues, and finally (3) the design of detection methods that can be used by performance tools to identify these missed opportunities in other applications. To guide our discussion, we group the performance opportunities we have discovered into four categories:

Unobvious missed parallelization opportunities in areas of the application where using the GPU would improve performance: What makes an unobvious region for conversion unobvious is the unknown benefit of converting the region to the GPU. The uncertainty of the conversion is caused by the assumption that the region does not have the necessary characteristics for profitable parallelization on the GPU. The characteristics needed are high parallelism, a flat memory structure (single dimensional arrays), and workload levels high enough to overcome the overheads associated with moving the computation to the GPU. Reducing the uncertainty of converting a region to the GPU is key to discovering unobvious parallelization opportunities. Reducing uncertainty requires that we identify CPU regions contributing significantly to runtime, determine the underlying memory structure of variables accessed within the region, and estimating the overheads of transferring work to/from the GPU.

We describe the issue of missed parallelizations and techniques to address them in more detail in Section III.

Duplicate data transfers causing unnecessary transfers of data already residing in physical memory on the CPU or GPU: The existence of unnecessary transfers is caused by the way GPU accelerated functionality is introduced into applications. The most common method of adding GPU functionality to existing applications is by dropping-in GPU replacements to CPU functionality. GPU replacements often taking the form of a ”GPU-ized” library (such as the use of accelerated libraries like cuFFT [29], CUSP [6], and others [9], [27]), a parallel code section inserted by the compiler (such as those generated by OpenACC [39]), or a block of user written code. Duplicate data transfers can occur when multiple replacements are in use by the application or when CPU-style behavior must be emulated to conform to the existing application structure. When multiple replacement libraries are in use, duplicate transfers to the GPU can occur because the replacements cannot communicate what data they have already moved to the GPU with one another. When CPU behavior must be emulated, the replacement library cannot assume that CPU data wont change between entrances to the library, requiring that all CPU data needed by GPU computation be transferred at every entry to the replacement library. The underlying cause of duplication is the lack of reuse of GPU resident memory and the assumption that data has been modified in-between calls to dropped-in replacements. A survey of large science applications conducted by Oak Ridge National Laboratory [17] lists the lack of GPU data reuse as one of the key performance issues faced by many high performance accelerated applications.

We describe the issue of duplicate data transfers and techniques to address them in more detail in Section IV.

Synchronizations between the CPU and GPU that are unnecessary or performed before needed, reducing CPU - GPU computation overlap. Misplaced or unnecessary synchronization occur when a synchronous operation happens before data is actually needed by the CPU or GPU. The existence of synchronization errors is typically due to the drop-in replacement method used by applications, such as by usage of a ”GPU-ized” library. Dropped-in replacements are typically required to emulate CPU-style behavior to operate within existing application structures. A requirement of emulating CPU-style behavior is ensuring that the results of a GPU computation are in CPU memory before returning to the application framework, requiring a synchronous memory transfer upon exit of the library. However, applications may not need the GPU data immediately on exit of the library (or even at all) making the synchronous operation unnecessary.

We describe synchronization issues and techniques to identify misplaced and unnecessary synchronizations in more detail in Section V-A.

Unnecessary Just-In-Time (JIT) compilation of GPU code on every execution of the application, increasing the overhead of using a GPU within the application. JIT compilation occurs when the application contains native GPU code that is incompatible with the GPU in use on the system [28]. The incompatibility is the result of specifying the incorrect GPU architecture at compile time or requiring the code to be generated from virtual code by the GPU device driver during execution. When an application is compiled for an incompatible architecture, application performance is affected due to the cost of performing the JIT compilation and by GPU code inefficiencies introduced by selecting the wrong virtual architecture at compile time. The effect in HPC environments can be magnified because the JIT-generated native code is not cached for subsequent executions. In addition, if the default virtual architecture targeted by the compiler is not a good match for the devices actually in use on the system, then the code may not be able to efficiently exploit to the GPU. When these easily correctable inefficiencies exist in the application, *no* notice is given to the user that performance is being negatively affected.

We describe the JIT compilation issue and techniques to address them in more detail in Section VI.

In Section II, we discuss currently available tools for detecting performance issues in GPU applications and why these tools do not provide the information necessary to fully diagnose performance issues in these categories. We discuss the performance challenges of missed unobvious parallelization opportunities, duplicate data transfers, explicit and implicit synchronizations, and JIT compilation in Sections III, IV, V, and VI respectively.

II. Related Work

Performance tools have been a key area of research since the introduction of the first multiprocessing machine. The introduction of accelerator computation has brought a new emphasis on the development of tools to find ways to improve application performance. Research initially started on developing tools to improve performance of application code already running on the GPU. GPU profiling and tracing tools were developed to detect GPU idleness [10], [20], [22], [23], [30], CPU idleness waiting on GPU completion [10], [20], [22], [30], warp occupancy [10], [22], [30], cache behavior [22], [30], instrumentation of GPU code [13], on-device synchronization issues [10], [30], and workload balance between accelerators on the same node [8]. These approaches have been beneficial in showing application developers how to improve the performance of code already written for GPUs.

The focus of our research is not on improving the effeiciency of GPU code, but on improving whole application performance by looking for performance opportunities outside of the GPU. Recently, the focus of tools has shifted to looking for performance opportunities outside of the GPU. Techniques to detect missed parallelization opportunities [4], [5], [7], [18], [26], [31] and detecting synchronization issues [2] have been developed to improve whole application performance. We discuss these contributions below.

A. Detection Approaches

Existing detection approaches have attempted to identify missed parallelization opportunities [4], [5], [7], [18], [26], [31], duplicate data transfers [30], and synchronization issues [2]. The approaches used to identify missed parallelization opportunities are:

Pattern-based approaches identify parallelizable code sections by comparing arithmetic operation, control flow, and memory access patterns in an application to a set of known parallelizable patterns [7], [31]. Static analysis is performed on a compiler-generated intermediate representation of the application to uncover the operations, control flow, and the memory access patterns contained within. If a pattern within the application matches a set of known parallelizable patterns, that code section is considered to be suitable for GPU parallelization. Our techniques use dynamically obtained information to detect parallelizable patterns that can be hidden from static analysis techniques, such as the identification of a sequential memory access patterns that is hidden from static analysis because it occurs in memory reached by a multi-level pointer access.

Algorithm classification methods [4], [18], [26] take a similar approach to pattern-based methods to identify parallelism. Algorithm classification approaches compare the algorithms in use by the application (such as a matrix multiplication or an arithmetic operation between vectors) to a list of known parallelizable algorithms. Algorithm classification approaches require that a developer manually specify the algorithm types used in a for loop to create a prediction.

A machine learning approach [5] that identifies areas of parallelization by using a machine learning model to estimate GPU performance of each individual loop in an application. Using static and dynamic analyses, a set of program properties is obtained that are used as input for a machine learning model. The program properties gathered include the number of instructions, number of memory/control/integer operations, the number of loop independent operations, and a set of features detailing the memory access pattern of the loop. The output of the model is an estimate of the computational speed-up that would be achieved by converting the loop to the GPU. A benchmark suite containing a pure CPU and hybrid CPU/GPU implementations of each benchmark was used to evaluate the machine learning method. The benchmarks used to train and evaluate the model consisted of small test applications, with source code length measured in the thousands of lines of code, performing a single task in isolation. The machine learning method was used to create a prediction based on the CPU version of the benchmark, with the hybrid CPU/GPU implementation of the same benchmark used to assess the accuracy of the prediction. These experiments showed that their prediction was within 22% of the actual speedup. However, the machine learning approach was not attempted on real world applications that typically are composed of a number of different types of tasks and number in the tens to hundreds of thousands of lines of code

The machine learning approach predicted only computational speedup of a loop and does not take into account overhead such as data transfer costs. The target of their work was identifying areas where high computational speedup (>4X performance) could be obtained. We are focused on parallelization of areas that may have lower computational speed-up where data transfer costs can dramatically effect the outcome in terms of reducing whole application runtime. We focus on low speed-up areas for two reason. First, they are the most likely to be passed over by application developers when implementing GPU parallelism into their applications. Second, these can result in significant reductions in runtime due to their presence on the critical paths of the application.

Researchers have used *blame analysis* to manually diagnose the presence of synchronization issues in applications. Blame analysis is a feature, first introduced in HPCToolkit [2] and later adopted by nvprof as *dependency analysis* [30], associating the ”blame” for synchronization delays with the device that is responsible for the delay. For example, if the CPU is waiting for the GPU to complete, the blame for the time spent waiting is placed on the GPU kernels active within the GPU. Blame analysis gives a developer an idea that there may be an issue with data transfer and synchronization operations by blaming a large percentage of total runtime on those operations. However, blame analysis does not give the developer any information on whether or not the transfers are necessary or if the synchronization operations can be moved to reduce delay. Our approach is designed to give developers information on the necessity of synchronization and transfer operations including where to place these operations to improve efficiency.

III. Unobvious Parallelization Opportunities

Unobvious parallelization opportunities exist in applications primarily for two reasons: (1) they have source code structures that do not appear to be favorable to parallelization and (2) they appear to have a minor benefit or even negative performance impact on application performance.

An example of one of these missed unobvious parallelization opportunities can be seen in the code excerpt taken from the 208K line molecular dynamics application LAMMPs [34] from Sandia National Laboratory (shown in Figure 1). This code from LAMMPs shows characteristics that are bad for GPUs: unknown number of loop iterations, multiple multi-level pointer reads and writes, and the presence of a branch condition. When you look at the code, it appears that the

```
for (int i = 0; i < nlocal; i++) {
    if (mask[i] & groupbit) {
      double dtfm;
      dtfm = dtf / mass[type[i]];
      v[i][0] += dtfm * f[i][0];
      v[i][1] += dtfm * f[i][1];
      v[i][2] += dtfm * f[i][2];
  }}
```

Fig. 1: Example of a missed parallelization opportunity from LAMMPs

CPU-to-GPU memory transfers for v[i] and f[i] need to be done in separate short (three elements at a time for each loop iteration) transfers, resulting in an inherently expensive pattern of transfers. So, the amount of work sent to the GPU may be too small to overcome the overhead of the multiple data transfers. It appears from the description given for existing techniques for detecting parallelization opportunities [4], [7], [18], [26], [31], that these techniques would make the same assumptions that developers would for this code region. Our goal is to test these research tools to verify our assessment of their techniques, however none are publicly available, and have never been tested on real-world code bases of this size.

We were able to obtain a 10% improvement to total application runtime by migrating the code in Figure 1 to the GPU. Previous techniques make inaccurate assumptions about variables `v` and `f` and the unknown value of `nlocal`. The assumption that can be drawn from source code is that `v[i]` and `f[i]` point to completely disjoint memory regions for every value of `i`. That assumption is wrong; `v` and `f` are each allocated in a contiguous manner where all indices's point into the same contiguous memory region. Thus the accesses at `v` and `f` can be rewritten as a single dimensional index. The contiguous allocation of `v` and `f` is hidden behind the memory management structure of LAMMPs. It would not be apparent to a developer that these variables are contiguous in memory without in-depth knowledge of the memory management framework in use. The unknown and possibly changing value of `nlocal` adds additional uncertainty since the loop may not operate long enough for any reasonable benefit to be achieved. In our experiments with LAMMPs, we found that the value in `nlocal` was high (over 400,000).

Approaches to detect missed parallelization opportunities need to be able to reveal information about the actual memory access pattern in use and the length of time spent executing within these code segments.

A. Detecting Unobvious Parallelization Opportunities

The behavior of long running loops with sequential memory access patterns indicates the presence of a loop that is favorable to conversion to the GPU. We view long running loops as GPU favorable because the computation is likely to run long enough to outweigh the overheads associated with GPU computation, such as memory transfer time and the latency of launching the kernel. Loops with only a small amount of execution time on the CPU may have overhead that outweighs any computational benefit, so we consider these to be *unlikely* candidates for conversion. A sequential memory access pattern

```
...
 for(int i = 0; i < n; i++)
    fftw_execute_dft(plan, &data[i],
                           &data[i]);
...
```

A: *Excerpt invoking cuFFT*

```
void fftw_execute_dft(plan, in, out){
    ...
    cudaMemcpy(in, dev);
    [Compute FFT on GPU]
    cudaMemcpy(dev, out);
    ...
}
```

B: *cuFFT library code for function fftw_execute_dft*

Fig. 2: QBox and Qball's usage of Nvidia's cuFFT library to accelerate discrete Fourier transform calculations

is often favorable because it allows the GPU to combine memory operations by different threads into a single memory transaction. GPUs are well-suited to codes with high memory bandwidth requirements [12], [25], so identifying codes with this characteristic indicates GPU favorability.

We designed a technique to identify these behaviors in applications using a dynamic approach combining CPU profiling with memory tracing. The first step uses CPU profiling to obtain information about the execution time of loops within the application. A loop is considered for conversion if it constitutes a large enough fraction of application execution time to be worth the effort of conversion. We can leverage existing performance profilers [2], [20], [33], [36] to accomplish this since they collect the needed CPU profiling information already. The second step uses memory tracing to determine if the loop under consideration has a memory access pattern suitable for parallelization. For each candidate loop, memory tracing is performed on a single representative instance of the candidate loop. Instrumentation inserted into the loop records the addresses used by all load and store operations. A separate memory tracing run of the application would be used to collect traces so profiling results are not perturb. We determine the favorability of the loop to parallelization by analyzing the memory access patterns contained in the trace, looking for contiguous ranges of virtual memory addresses accessed during loop execution. If contiguous virtual memory address ranges can be formed from the individual virtual memory addresses captured, the loop is identify as containing a sequential memory access pattern suitable for the GPU parallelization. Loops identified by both performance profiling and memory tracing as being suitable for the GPU would be marked as a missed unobvious parallelization opportunity.

IV. Duplicate data transfers

Duplicate data transfers are unnecessary transfers of data between CPU and GPU memory. A transfer is unnecessary if the data already exists in the memory space to which it is being written. Unnecessary transfers occur when developers cannot make assumptions about data modifications between regions of code, such as between functions or libraries, within the application. A region processing data using the GPU may conservatively decide to re-transfer data already resident on the GPU if it could have been modified by another region. Typically, unnecessary transfers occur when libraries are used to add GPU acceleration to applications or when multiple dropped-in GPU replacements to CPU functionality are introduced into an application.

Figure 2 shows an example of an unnecessary transfer from the 100K line QBox [16] molecular dynamics application developed at U.C. Davis. The unnecessary transfer is caused by QBox's usage of the discrete Fourier transform library, cuFFT [29]. cuFFT is a library developed by Nvidia as a drop in replacement for the CPU discrete Fourier transform library, FFTW [14]. Maintaining compatibility with FFTW requires that all of the steps needed to setup the transform on the GPU, such as transferring data, must be done within the cuFFT library itself. In the example shown in Figure 2, QBox is performing a Fourier transform on data starting at location `data[i]` where `data` is a flat single dimensional array. cuFFT transfers `N` elements starting at position `data[i]` to the GPU. `N` is defined by the application on initialization of the FFT library and is stored in the variable `plan`. The transform is computed on the GPU and the results are transferred back to `data[i]`. Since the values located within `data` are not modified between each subsequent transform, each transfer after the initial iteration contains duplicated data that does not need to be transferred. The duplicate transfers increase application runtime by approximately 40%.

The issue present in Qbox shown in Figure 2 extends to QBall [11], an enhanced version of QBox created by Lawrence Livermore National Laboratory. QBall contains experimental features, such as support for f-projectors and the implementation of a highly-scalable algorithm to calculate the time-dependent Density Functional Theory on a many-body system. QBall inherits its application structure, including the structure of the FFT computation, from QBox. By inheriting QBox's FFT structure, QBall also inherited the performance issue seen in QBox when linked with cuFFT. The same performance issue described above for QBox shown in Figure 2 appears in QBall. The duplicate transfers seen in QBall increase application runtime by the same amount, approximately 40%.

A. Detection

We developed a content-based data deduplication approach to identify duplicate transfers. Content based data deduplication approaches compare the hash values of data regions to identify duplicates [35], [37]. Our implementation intercepts calls to cudaMemcpy (and its derivatives such as cudaMemcpyAsync) to obtain the location of data being transferred between the CPU and GPU. If a match to a previous transfer is detected, a stack trace at the location of the duplicate transfer is recorded. Intercepting the data transfers between the CPU and GPU is simplified by the need to use standard calls to invoke the transfers.

To further evaluate our automation of the detection of duplicate transfers, we created a prototype tool implementing our content based data deduplication approach. Using this prototype tool, we detected that approximately 70% of the data transfers that take place in the deep neural network framework Tensorflow [1] contain duplicate data. Currently, we are in the process of determining what impact the duplicate transfers have on Tensorflow's execution time and how to eliminate these transfers.

With a small extension to our detection technique, we would also be able to automatically correct duplicate transfers as they occur. We still must address the fact that the duplicate transfers we identify are not guaranteed to be duplicates on subsequent runs of the application with different inputs. To overcome this limitation, permanent instrumentation will be inserted at data transfers containing duplicate data to always perform a hash check before the transfer. If a transfer that we expect to be a duplicate is not, we perform the transfer and record a stack trace to alert the user. This approach relies on the ability to generate a hash of the data in a transfer request faster than the transfer could take place. The time cost of performing a data transfer can be decomposed into startup costs, the time it takes to move the first byte of data, and the per byte transfer cost after startup. GPU data transfers have high startup costs but low per byte data transfer costs [15] while hash checking has very low startup costs with higher per-byte data costs than a GPU transfer. The fastest CPU hashing algorithm we have tested so far, xxHash [24], adds approximately 3% overhead to application execution time on the applications we have tested so far. Using a hashing approach to identify and eliminate duplicate transfers in QBox [16] we can achieve an estimated 16 - 35% of the benefit we obtained via manual tuning.

V. SYNCHRONIZATIONS

There are two types of GPU synchronizations operations that we have identified: implicit and explicit. Implicit synchronizations are caused by side effects of operations such as memory transfers or allocations. Explicit synchronizations are manually invoked by the application to synchronize the CPU with the GPU. When a synchronization takes place, the CPU waits for the GPU to complete all existing operations before continuing. First, we want to remove an unnecessary or redundant synchronization operation. Second, we want to delay for as long as possible any operation requiring a synchronization with the GPU to maximize CPU - GPU computational overlap. Ideally, the point where an application performs a synchronization operation is right before the result of the operation is needed by the CPU. We discuss how synchronization errors present themselves in applications and describe an automated method to detect synchronization issues.

A. Implicit Synchronization Issues

Implicit synchronizations occur when a library call made by an application synchronizes with the GPU before returning control. The most typical implicit synchronization operations are synchronous data transfers and memory allocation requests. The challenge developers face is determining how to delay (or replace) operations that implicitly synchronize. The problem of avoiding implicit synchronization is made more challenging when the synchronization is hidden from application code, such as when a library in use by the application is itself making an implicit synchronization call.

The interaction between QBox/QBall with cuFFT, shown in Figure 2, is an example of an implicit synchronization. Figure 2B shows cuFFT making two calls to cudaMemcpy where each call to cudaMemcpy performs an implicit synchronization. However the result from the second cudaMemcpy operation is not used until after the for-loop in Figure 2A. The result of these unnecessary (and early) implicit synchronizations used by cudaMemcpy is an increase in application execution time by 40%. The cumulative effect of removing duplicate data transfers and implicit synchronizations from both QBox and QBall was a reduction in execution time by 85%.

In cuIBM [21], a 17K line computational fluid dynamics simulator from George Washington University, the implicit synchronization operations of cudaMalloc and cudaFree delay CPU execution unnecessarily. The cudaMalloc and cudaFree operations take place on the creation and destruction of temporary GPU vectors. A vector in cuIBM would be created (causing a synchronization), filled with data via an asynchronous memory transfer, used by GPU computation, and then in most cases would be destroyed (causing another synchronization). This pattern of creating and destroying temporary memory spaces for vectors is common throughout the execution of cuIBM. The result is an unnecessary delay of CPU code not dependent on calculations from the GPU. We corrected the problem by allocating vectors that would be reused only once. The result was a reduction in cuIBM's execution time by 8%.

B. Excessive Explicit Synchronizations

Explicit synchronizations are used to wait for the completion of in-progress asynchronous operations such as data transfers. The challenge that developers face is determining when a explicit synchronization is necessary and where to place it. When a developer does this incorrectly, application performance can be reduced significantly.

In Hoomd [3], a 112K line molecular dynamics simulator, a removal of an explicit synchronization operation reduced execution time by 37%. The explicit synchronization, shown in Figure 3, is used to wait for the GPU to update the shared variable `sharedStatus`. `sharedStatus` indicates whether the GPU computation failed because not enough GPU memory was allocated for the operation. The value of `sharedStatus` is true (successful GPU completion) for every iteration of the for-loop except the first iteration when GPU memory is initially allocated by the CPU. Even though the value of `sharedStatus` is true for iterations `2 to N` of the for-loop, the application still synchronizes with the GPU on every iteration causing the reduction in performance by delaying the unrelated CPU computation.

(a) Original version of Hoomd's main compuational loop

```
// Status variables shared between CPU and GPU
bool sharedStatus; int * GPUData;
// Size of GPUData
int size = 0;
cudaMalloc(&GPUData, size);
// Main computational loop of hoomd
for(step = 0; step < nsteps; step++) {
    ...
    do {
        GPUComputation<<< >>>(sharedStatus,
                              GPUData,
                              size,
                              ...);
        // Synchronize to get GPU updates
        // to sharedStatus
        cudaDeviceSynchronize();
        // If sharedStatus is false...
        // allocate more GPU memory and retry
        if(sharedStatus == false) {
            size = len(GPUData) + ...;
            cudaMalloc(&GPUData, size);
        }
    } while(sharedStatus == false);
    // CPU work not dependant on GPU data
    for(i = 0; i < count; i++)
        ...
    ...
    // Existing Implicit Synchronization
    cudaMemcpy(...)
    ...
}
```

Red statements depend on results from GPU

(b) Manually improved version with reduced CPU delay

```
// Status variables shared between CPU and GPU
bool sharedStatus; int * GPUData;
// Size of GPUData
int size = MAX_DATA_SIZE;
cudaMalloc(&GPUData, size);
// Main computational loop of hoomd
for(step = 0; step < nsteps; step++) {
    ...

        GPUComputation<<< >>>(sharedStatus,
                              GPUData,
                              size,
                              ...);










    // CPU work not dependant on GPU data
    for(i = 0; i < count; i++)
        ...
    ...
    // Existing Implicit Synchronization
    cudaMemcpy(...);
    ...
}
```

Blue statements have no GPU dependencies

Fig. 3: A flat representation of an explicit synchronization error in the main computational loop in Hoomd.

C. Dectecting Implicit and Explicit Synchronization Opportunities

A synchronization opportunity is present when a synchronization operation causes unnecessary delay. We view delay as unnecessary when CPU computation blocks for the GPU but does not access data shared with the GPU. The result of CPU computation being delayed unnecessarily is a reduction in CPU - GPU overlap. We have identified three types of unnecessary delay: (1) when the CPU does not access shared data (data shared with the GPU) after the synchronization, (2) when the placement of the synchronization is far from the first access of shared data by the CPU, and (3) when CPU computation not dependent on GPU data is delayed by a synchronization. Unnecessary delay can be reduced by removing the synchronization, moving the synchronization closer to shared data access, or by reordering CPU computation to make the synchronization obsolete.

Figure 4 shows an example of how delaying (or removing) a synchronization can reduce the amount of time the CPU is delayed. This is a general example representing the behavior seen in QBox, cuIBM, and Hoomd. In the code section shown in the left of Figure 4, a synchronization operation occurs after a memory transfer even though the shared data (stored in `dest`) may not be accessed. If `cond1` is true, the synchronization is unnecessary, blocking the CPU for no reason. In the other cases, there may be enough CPU work performed before the access to `dest` that a delay could be avoided by moving the synchronization closer to the shared data access. The impact of the unnecessary delay can be magnified if this code section is called multiple times, such as within a loop.

Our technique to identify unnecessary synchronizations uses a dynamic approach combining the techniques of profiling, memory tracing, and program slicing [19], [38] to identify where a synchronization opportunity is located and how to correct its behavior. Our approach can be broken down into five steps: (1) identify the synchronization operations that cause long delays on the CPU, (2) determine what data is shared between the CPU and GPU, (3) identify the CPU instructions that access shared data, (4) determine how far the instruction performing the synchronization is from the first CPU instruction that accesses data shared, and (5) determine if CPU computation exists that does not depend on shared data. Using this information, we will generate the corrective measure that should be taken and an estimate of the amount of time that could be saved if the measure was taken. We explain how to gather this information below.

We target synchronization operations with long CPU delays because a change in their synchronization behavior can result in significant improvements in CPU - GPU overlap. Existing CPU performance profiling tools already obtain the location of a synchronization and how long the CPU blocks at the

(a) Original version with synchronizing too early

```
cudaMemcpyAsync(dest, src, len, ...);
...
cudaDeviceSynchronization();
v = 1;
if (cond1) {
  ... // A lot of CPU computation
} else if (cond2) {

  v = dest[0];
  ... // Any amount of CPU computation
} else {
  ... // A lot of CPU computation

  v = dest[1];
}
result = v;
```

(b) Improved version with reduced CPU delay

```
cudaMemcpyAsync(dest, src, len, ...);
...

v = 1;
if (cond1) {
  ... // A lot of CPU computation
} else if (cond2) {
  cudaDeviceSynchronization();
  v = dest[0];
  ... // Any amount of CPU computation
} else {
  ... // A lot of CPU computation
  cudaDeviceSynchronization();
  v = dest[1];
}
result = v;
```

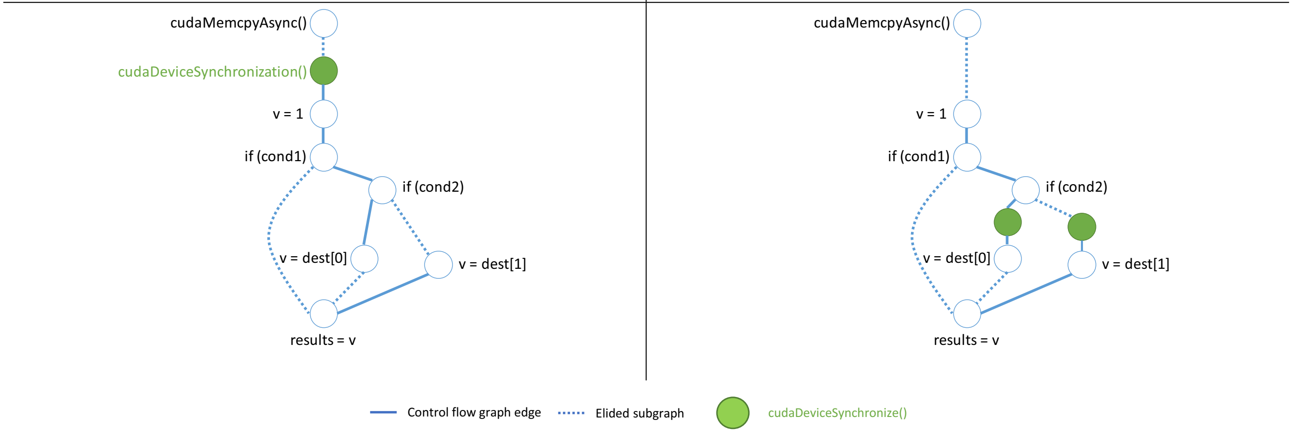


Fig. 4: Illustrative example of an early synchronization causing unnecessary delay

synchronization [2], [10], [20], [22], [30]. We use synchronization delay information obtained by one of these profilers to determine the synchronization operations with long delays that we will perform further analysis on.

At each synchronization, we must identify what data is shared between the CPU and GPU. Data can be shared between the CPU and GPU using one of two methods: a memory transfer or through the mapping of CPU memory pages to the GPU. Both methods are initiated through requests made through a standardized API. We identify data being shared between the CPU and GPU by intercepting these requests and recording the memory location (and size) of the data being shared.

We identify where shared data is accessed by CPU computation using a combination of source code analysis and memory tracing. We first identify the instructions containing pointers that might point to shared data. With this set of instructions, we use memory tracing to determine the instructions that access a memory location containing shared data at runtime.

The ordered set of instructions accessing shared data allows us to identify two types of unnecessary delay: no shared data access by the CPU and synchronization far from the use of shared data. If the set of instructions accessing shared data is empty, no access to shared data occurs and the synchronization can be removed. If the total number of instructions between the end of the synchronization and the first access to shared data is large, we know that CPU delay could be reduced by moving the synchronization closer to this access. The corrective measure is to move the synchronization to the location of the access. The synchronization opportunities in QBox [16], QBall [11], and cuIBM [21] fall under these types and are identified here.

The approach that we use will detect shared data usage down the most common path, but will not detect possible paths after the synchronization where shared data uses may occur. For example, while we are monitoring the application in Figure 4, if the path reaching `v = dest[0]` is not traveled, we must ensure that the application remains correct if the path is ever taken. We propose to resolve correctness down unseen paths by using static analysis to identify all untraversed paths in the control flow graph that follow the original location of the synchronization. Conservatively, a location before the first memory reference on the untraversed path is where a synchronization has to occur.

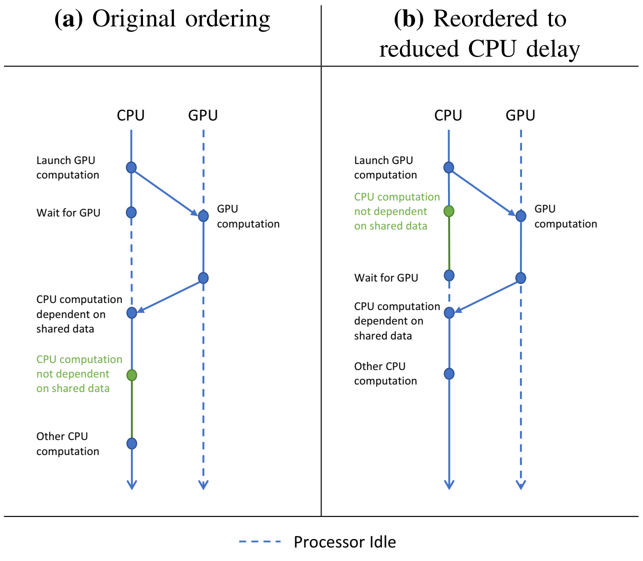
The third type of unnecessary delay, illustrated in Figure 5, is when CPU computation not dependent on GPU data is being delayed by a synchronization. A CPU computation is not dependent on GPU data if the values of variables used in the computation are not affected by changes to data shared with the GPU. The delay is unnecessary because it can be reduced by performing the CPU computation before the synchronization, increasing CPU - GPU overlap. Identifying this case of unnecessary delay requires that we locate instructions that do not depend on GPU data. We use program slicing [19], [38] to identify instructions that do not depend on GPU data. An instruction is dependent on GPU data if the values used by the instruction are affected by data shared with the GPU. A forward slice is created starting at the synchronization operation with the locations of shared data being used as the criterion for the instructions to be included in the slice. The result is a slice containing the instructions that *may depend* on data shared with the GPU. We are interested in the set of instructions that are *not* in the slice since they do not depend on data shared with the GPU. If the number of instructions not in the slice is large (say, greater than a few hundred instructions), then moving these instructions before the synchronization operation could have a noticeable benefit. The synchronization opportunity in Hoomd [3] falls under this type of unnecessary delay and would be detected here.

Our current work is focused on automating the identification of instructions accessing shared data and determining if CPU computation exists after a synchronization that does not depend on values stored in shared data. We will use binary code instrumentation to identify the instructions that access shared data by instrumenting the load and store requests made by the CPU after a synchronization call is made. Dyninst [32], a binary code instrumentation and analysis toolkit, will capture the addresses used by individual load and store instructions between the end of the synchronization and the first instruction accessing shared data. Using the information obtained during instrumentation, we will modify existing automated program slicing techniques to identify the instructions not dependent on shared data.

VI. JIT Compilation

GPU native code may need to be generated, from a GPU virtual architecture at runtime because there is no native GPU code present in the executable file, or the code that is present is for the wrong model GPU. The lack of native GPU support is a product of the misconfiguration of the applications at compile time. Common reasons for misconfiguration are a developer not knowing the correct native architecture of the GPU, build systems such as CMake incorrectly identifying the native architecture, and use of compiler defaults that produce incompatible binaries for most GPUs. When a missconfiguration of the architecture occurs, application users are not notified that their application is misconfigured, either at compile or execution time.

The cuIBM [21] application is an example of the impact a misconfiguration can have on performance. 18% of cuIBM's execution time is spent performing JIT compilation because the wrong architecture is selected by the build system. cuIBM defaults to compiling to the virtual architecture ”compute_20”, while the GPUs in the system actually support ”compute_35”.



(a) Original ordering
(b) Reordered to reduced CPU delay
CPU
GPU
Launch GPU computation
Wait for GPU
GPU computation
CPU computation dependent on shared data
CPU computation not dependent on shared data
Other CPU computation
Processor Idle

Fig. 5: Illustrative example of unnecessary delay caused by delaying CPU computation not dependent on GPU data

Since we ran cuIBM in an HPC environment (the Cray Titan supercomputer at Oak Ridge), the JIT compilation is not cached and must be performed at every execution.

Application incompatible with its GPU codes seems to be quite simple and is surprising (but widely present). We can detect GPU code incompatibility at application startup and provide explicit instructions to the user as how to produce a more efficient executable.

VII. Conclusion

The increased parallelism offered by many-core architectures is difficult for developers to exploit. In response, performance tool and application developers created techniques that address some of these difficulties. Existing techniques primarily address difficulties in the areas of the identification of CPU code that may be suited for many-core parallelization and improving the efficiency of existing many-core code. Despite their efforts, some significant performance opportunities have remained. We identified four performance issues that have impacted several high performance scientific applications utilizing GPUs for computation: unobvious missed parallelization opportunities, duplicate data transfers, synchronization issues, and JIT compilation.

What links the issues together is the lack of performance tools and techniques to detect their presence. We have developed techniques that can detect when and where these issues are present within applications. These techniques use a combination of memory tracing, program slicing, content based data deduplication, and CPU profiling to detect their presence. When we applied these techniques to a set of high performance scientific applications, application execution time was reduced by 18% to 85%. We believe that the issues we have identified impact a wider range of applications than only the ones we have tested so far. Our current work is focused on automating these techniques to detect their presence to allow our techniques to be more easily employed by others.

References

[1] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng, "Tensorflow: A system for large-scale machine learning," in *the 12th USENIX Symposium on Operating Systems Design and Implementation*, ser. (OSDI 16), Savannah, GA, November 2016. [Online]. Available: https://www.usenix.org/system/files/conference/osdi16/osdi16-abadi.pdf

[2] L. Adhianto, S. Banerjee, M. Fagan, M. Krentel, G. Marin, J. Mellor-Crummey, and N. R. Tallent, "HPCToolkit: Tools for performance analysis of optimized parallel programs," *Concurrency and Computation: Practice and Experience*, vol. 22, no. 6, April 2010.

[3] J. A. Anderson, C. D. Lorenz, and A. Travesset, "General purpose molecular dynamics simulations fully implemented on graphics processing units," *Journal of Computational Physics*, vol. 227, no. 10, May 2008.

[4] J. Ansel, C. Chan, Y. L. Wong, M. Olszewski, Q. Zhao, A. Edelman, and S. Amarasinghe, "Petabricks: A language and compiler for algorithmic choice," in *the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. (PLDI '09), Dublin, Ireland, June 2009.

[5] N. Ardalani, C. Lestourgeon, K. Sankaralingam, and X. Zhu, "Cross-architecture performance prediction (XAPP) using CPU code to predict GPU performance," in *the 48th International Symposium on Microarchitecture*, ser. (MICRO), Waikiki, Hawaii, December 2015.

[6] N. Bell, S. Dalton, and L. N. Olson, "Exposing fine-grained parallelism in algebraic multigrid methods," *SIAM Journal on Scientific Computing*, vol. 34, no. 4, July 2012.

[7] L. Carrington, M. M. Tikir, C. Olschanowsky, M. Laurenzano, J. Peraza, A. Snavely, and S. Poole, "An idiom-finding tool for increasing productivity of accelerators," in *the 25th International Conference on Supercomputing*, ser. (SC '11), Tucson, Arizona, June 2011.

[8] L. Chen, O. Villa, S. Krishnamoorthy, and G. R. Gao, "Dynamic load balancing on single- and multi-GPU systems," in *the 2010 IEEE International Symposium on Parallel Distributed Processing*, ser. (IPDPS '10), Atlanta, GA, April 2010.

[9] S. Chetlur, C. Woolley, P. Vandermersch, J. Cohen, J. Tran, B. Catanzaro, and E. Shelhamer, "cuDNN: Efficient primitives for deep learning," *Computing Research Repository*, vol. abs/1410.0759, October 2014.

[10] B. R. Coutinho, G. L. M. Teodoro, R. S. Oliveira, D. O. G. Neto, and R. A. C. Ferreira, "Profiling general purpose GPU applications," in *the 21st International Symposium on Computer Architecture and High Performance Computing*, ser. (SBAC-PAD '09), Sao Paulo, Brazil, October 2009.

[11] E. W. Draeger, X. Andrade, J. A. Gunnels, A. Bhatele, A. Schleife, and A. A. Correa, "Massively parallel first-principles simulation of electron dynamics in materials," in *the 2016 International Parallel and Distributed Processing Symposium*, ser. (IPDPS '16), Chicago, IL, May 2016.

[12] Z. Fan, F. Qiu, A. Kaufman, and S. Yoakum-Stover, "GPU cluster for high performance computing," in *the 2004 ACM/IEEE Conference on Supercomputing*, ser. (SC '04), Pittsburgh, PA, November 2004. [Online]. Available: https://doi.org/10.1109/SC.2004.26

[13] N. Farooqui, A. Kerr, G. Eisenhauer, K. Schwan, and S. Yalamanchili, "Lynx: A dynamic instrumentation system for data-parallel applications on GPGPU architectures," in *the 2012 IEEE International Symposium on Performance Analysis of Systems Software*, ser. (ISPASS '12), New Brunswick, NJ, April 2012.

[14] M. Frigo and S. G. Johnson, "The design and implementation of FFTW3," *Proceedings of the IEEE*, vol. 93, no. 2, Febuary 2005.

[15] Y. Fujii, T. Azumi, N. Nishio, S. Kato, and M. Edahiro, "Data transfer matters for GPU computing," in *the 2013 International Conference on Parallel and Distributed Systems*, ser. (ICPADS '13), Seoul, South Korea, December 2013.

[16] F. Gygi, "Architecture of Qbox: A scalable first-principles molecular dynamics code," *IBM Journal of Research and Development*, vol. 52, no. 1, January 2008. [Online]. Available: http://dl.acm.org/citation.cfm?id=1375990.1376003

[17] W. Joubert, R. Archibald, M. Berrill, W. M. Brown, M. Eisenbach, R. Grout, J. Larkin, J. Levesque, B. Messer, M. Norman, B. Philip, R. Sankaran, A. Tharrington, and J. Turner, "Accelerated application development: The ORNL Titan experience," *Computers and Electrical Engineering*, vol. 46, August 2015. [Online]. Available: //www.sciencedirect.com/science/article/pii/S0045790615001366

[18] K. Keutzer, B. L. Massingill, T. G. Mattson, and B. A. Sanders, "A design pattern language for engineering (parallel) software: Merging the PLPP and OPL projects," in *the 2010 Workshop on Parallel Programming Patterns*, ser. (ParaPLoP '10), Carefree, Arizona, March 2010.

[19] A. Kiss, J. Jasz, G. Lehotai, and T. Gyimothy, "Interprocedural static slicing of binary executables," in *the 3rd IEEE International Workshop on Source Code Analysis and Manipulation*, ser. (SCAM '03), Amsterdam, NL, September 2003.

[20] A. Knüpfer, C. Rössel, D. a. Mey, S. Biersdorff, K. Diethelm, D. Eschweiler, M. Geimer, M. Gerndt, D. Lorenz, A. Malony, W. E. Nagel, Y. Oleynik, P. Philippen, P. Saviankou, D. Schmidl, S. Shende, R. Tschüter, M. Wagner, B. Wesarg, and F. Wolf, "Score-P: A joint performance measurement run-time infrastructure for Periscope, Scalasca, TAU, and Vampir," in *the 5th International Workshop on Parallel Tools for High Performance Computing*, Berlin, Heidelberg, September 2011.

[21] S. Layton, A. Krishnan, and L. A. Barba, "cuIBM - a GPU-accelerated immersed boundary method," in *the 23rd International Conference on Parallel Computational Fluid Dynamics*, ser. (ParCFD '11), Barcelona, Spain, May 2011.

[22] A. D. Malony, S. Biersdorff, S. Shende, H. Jagode, S. Tomov, G. Juckeland, R. Dietrich, D. Poole, and C. Lamb, "Parallel performance measurement of heterogeneous parallel systems with GPUs," in *the 2011 International Conference on Parallel Processing*, ser. (ICPP '11), Taipei City, Taiwan, September 2011.

[23] A. D. Malony, S. Biersdorff, W. Spear, and S. Mayanglambam, "An experimental approach to performance measurement of heterogeneous parallel applications using cuda," in *the 24th ACM International Conference on Supercomputing*, ser. (ICS '10). Tsukuba, Ibaraki, Japan: ACM, June 2010. [Online]. Available: http://doi.acm.org/10.1145/1810085.1810105

[24] Mathias Westerdahl, "xxhash - xxh64," http://www.xxhash.com/.

[25] J. Nickolls and W. J. Dally, "The GPU computing era," *IEEE Micro*, vol. 30, no. 2, March 2010.

[26] C. Nugteren and H. Corporaal, "A modular and parameterisable classification of algorithm," Eindhoven University of Technology, Eindhoven, Netherlands, Tech. Rep. ESR-2011-02, Febuary 2011.

[27] Nvidia, *The CUBLAS Library*, 8th ed., 2016.

[28] Nvidia, *CUDA Compiler Driver NVCC - Reference Guide*, 8th ed., 2016.

[29] Nvidia, *The Cuda FFT Library*, 8th ed., 2016.

[30] Nvidia, *The Nvidia CUDA Profiler Users' Guide*, 8th ed., 2016.

[31] C. Olschanowsky, A. Snavely, M. R. Meswani, and L. Carrington, "PIR: PMaC's idiom recognizer," in *the 39th International Conference on Parallel Processing Workshops*, ser. (ICPP '10), San Diego, CA, September 2010.

[32] Paradyn Project, "Dyninst: Putting the performance in high performance computing," http://www.dyninst.org/.

[33] P. Petersen, "Intel® parallel studio," in *Encyclopedia of Parallel Computing*, 2011.

[34] S. Plimpton, "Fast parallel algorithms for short-range molecular dynamics," *Journal of Computational Physics*, vol. 117, no. 1, March 1995. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S002199918571039X

[35] S. Quinlan and S. Dorward, "Venti: A new approach to archival storage," in *the 2002 Conference on File and Storage Technologies*, ser. (FAST '02), Monterey, CA, 2002. [Online]. Available: http://dl.acm.org/citation.cfm?id=645371.651321

[36] SoftBank Group, *Allinea Forge User Guide*, 7th ed., 2016.

[37] C. A. Waldspurger, "Memory resource management in VMware ESX server," in *the 5th symposium on Operating Systems Design and Implementation*, ser. (OSDI '02), Boston, MA, December 2002. [Online]. Available: http://doi.acm.org/10.1145/844128.844146

[38] M. Weiser, "Program slicing," in *the 5th International Conference on Software Engineering*, ser. (ICSE '81), San Diego, CA, March 1981. [Online]. Available: http://dl.acm.org/citation.cfm?id=800078.802557

[39] S. Wienke, P. Springer, C. Terboven, and D. an Mey, "OpenACC: First experiences with real-world applications," in *the 18th International Conference on Parallel Processing*, ser. (Euro-Par '12), August 2012.