Exposing Hidden Performance Opportunities in High Performance GPU Applications

Benjamin Welton and Barton P. Miller

Computer Sciences Department

University of Wisconsin – Madison

Analysis of Paper by:
Chloe Frame
Chemical Engineering Graduate Student
Vanderbilt University

Outline

1

Objective of Paper

Introduction

What is the problem?

Why is this relevant?

2

Optimizations to GPU Performance

Unobvious Parallelization Opportunities

Duplicate Data Transfers

Synchronizations

3

Conclusions

How does this help us?

4

Comments

Personal opinions

Why did I choose this paper?

5

References

El-Araby, E. (2018). Exposing Hidden Performance Opportunities in High Performance GPU Applications. 2018 18TH IEEE/ACM INTERNATIONAL SYMPOSIUM ON CLUSTER, CLOUD AND GRID COMPUTING, 301–310.

Objectives

- The addition of accelerators, GPUs, have the potential to increase performance of applications
- Being able to effectively exploit parallelism requires developers of the applications to identify where accelerator parallelization is needed and ensuring the efficient interaction between CPU and GPU
- In this paper, significant untapped performance opportunities exist in these areas even in well-known, heavily optimized, real world applications created by experienced GPU developers

Objectives

The Problem

- These untapped performance opportunities exist because:
  - Accelerated libraries can create unexpected synchronization delay and memory transfer requests
  - Interaction between accelerated libraries cause unexpected inefficiencies when combined
  - Vectorization opportunities can be hidden by the structure of the program

Objectives

The Relevance

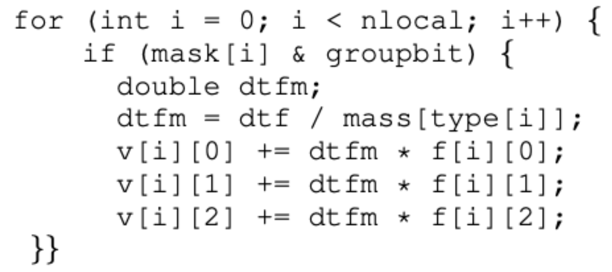
- The following commonly used applications were studied:
  - Qball
  - Qbox
  - cuIBM
  - Hoomd-blue
  - LAMMPs
- The focus of the research is not on improving the efficiency of GPU code, but on improving whole application performance by looking for performance opportunities outside of the GPU by using existing performance analyzers
- The goal:
  - Characterize missed performance opportunities in many-core applications and why they are difficult to identify
  - Show the performance benefit of correcting these issues
  - Illustrate the design of detection methods used by performance tools to identify missed opportunities in other applications
- By examining the unobvious missed parallelization opportunities, duplicate data transfers, and synchronization issues, GPU performance can be analyzed to highlight the needs for improvement

Unobvious Missed Parallelization Opportunities Overview

- What makes an unobvious region for conversion unobvious is the unknown benefit of converting the region to the GPU
- Necessary Characteristics:
  - High Parallelism
  - Flat memory structure (single dimensional array)
  - Workload levels high enough to overcome overhead of moving computation to GPU
- Need to understand where:
  - CPU regions contribute significantly to run-time
  - Underlying memory structure of variables is accessed within the region
  - Overheads of transferring work to and from the GPU

Unobvious Missed Parallelization Opportunities Example

- Example taken from LAMMPs showing unknown number of loop iterations
- It appears that CPU-to-GPU memory transfers for v[i] and f[i] need to be done in separate short transfers, three elements at a time for each loop iteration (very expensive pattern of transfers)
- The amount of work sent to the GPU may be too small to overcome the overhead of the multiple data transfers
- Wrong Assumption: The code is written thinking that v[i] and f[i] point to disjoint memory regions for every value of i
- Solution: The accesses at v and f can be rewritten as a single dimensional index
- Why? It’s not apparent unless you understand the memory management framework underneath that the allocation of v and f is hidden behind the memory management structure of LAMMPs



```
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

Unobvious Missed Parallelization Opportunities Enhancement

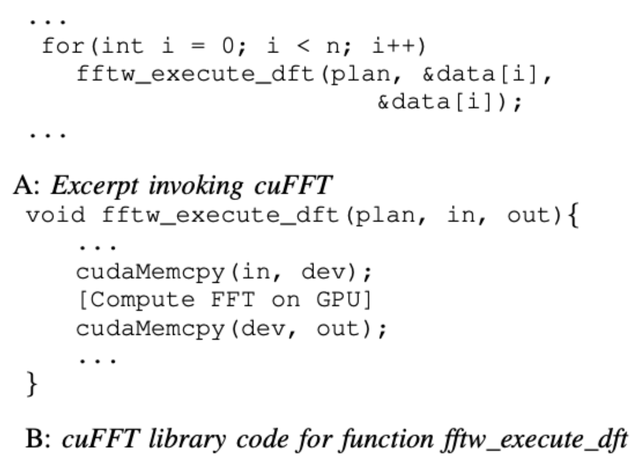
- A technique was designed to identify behaviors in applications using a combination of CPU profiling and memory tracing
- Step 1: CPU profiling is used to find the execution time of loops within the application
  - A loop is considered for conversion if it constitutes a large enough fraction of application execution time to be worth the effort of conversion. Using existing performance profilers (HPCToolkit, Score-P, Intel parallel studio, Allinea Forge) they collect the CPU profiling information
- Step 2: Memory tracing is used to determine if the loop under consideration has a memory access pattern suitable for parallelization
  - For each loop, memory tracing is performed on the loop. Instrumentation inserted into the loop records the addresses used by all load/store operations
  - The favorability of the loop to parallelization is analyzed by the memory access patterns contained in the trace, which looks for addresses accessed during loop execution
  - If virtual memory address ranges are formed from the individual virtual memory addresses, the loop is identified as containing a sequential memory access pattern suitable for 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

Duplicate Data Transfers
Overview

- The unnecessary transfer of data already residing in physical memory on the CPU or GPU
- The most common method of adding GPU functionality to existing applications is by adding GPU replacements to CPU functionality
- GPU additions often are:
  - libraries (e.g. cuFFT, CUSP, etc.)
  - a parallel code section inserted by the compiler (such as those generated by OpenACC)
  - a block of user written code
- Duplicate data transfers occur when multiple additions are in use by the application
- 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

Duplicate Data Transfers
Example

- Unnecessary transfers occur when developers cannot make assumptions about data modifications between regions of code, such as between functions or libraries, within the application
- Figure 2 shows an example of an unnecessary transfer from the 100K line QBox molecular dynamics application
- The unnecessary transfer is caused by QBox's usage of the discrete Fourier transform library, cuFFT which is a replacement for the CPU Fourier transform library, FFTW
- Steps:
  - Maintaining compatibility with FFTW requires that all steps needed to setup the transform on the GPU (transferring data), must be done using the GPU library
  - 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 and transfers N elements 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 <u>does not need to be transferred</u>
- The duplicate transfers increase application runtime by approximately 40%!



```
...
 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

Duplicate Data Transfers
Enhancement

- A content-based data deduplication approach to identify duplicate transfers was created to compare hash values of data regions to identify duplicates
- Process:
  - The implementation intercepts calls to cudaMemcpy 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 transfers
- Being able to continue this research would move towards the goal of being able to automatically correct duplicate transfers as they occur

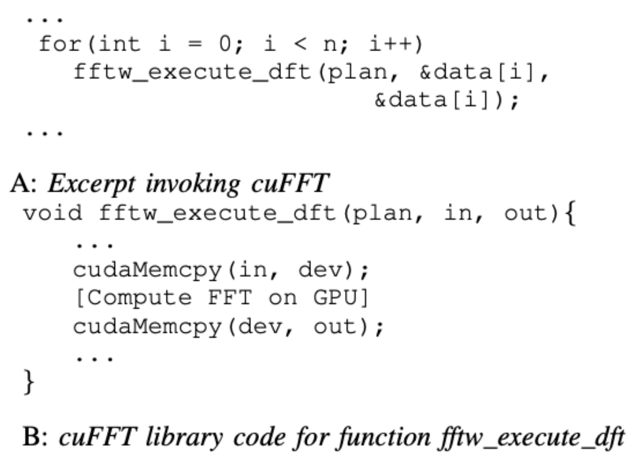
Synchronizations Overview

- Synchronizations between CPU and GPU that are unnecessary or performed before needed, reducing CPU - GPU computation overlap
- Errors in synchronization occur when a sync operation happens before data is needed by the CPU/GPU due to the drop-in replacement method used by applications, such as by usage of a GPU library
- Dropped-in replacements are required to emulate CPU-style behavior to operate within existing application structures ensuring that the results of a GPU computation are in CPU memory before returning to the application framework, requiring a sync memory transfer upon exit of the library
- First, we want to remove an unnecessary or redundant sync operations
- Second, we want to delay for as long as possible any operation requiring a synchronization with the GPU to maximize CPU - GPU computational overlap
- 2 Types of GPU synchronizations: implicit and explicit

Synchronizations

Implicit

- Implicit synchronizations are results of operations like as memory transfers or allocations
- Figure 2 also is an example of an implicit synchronization as well as duplicate data
- 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 execution time increased by 40% as a result of the unnecessary early implicit sync used by cudaMemcpy
- The cumulative effect of removing duplicate data transfers and implicit synchronizations from QBox was a reduction in execution time by 85%



```
...
 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);
    ...
}
```

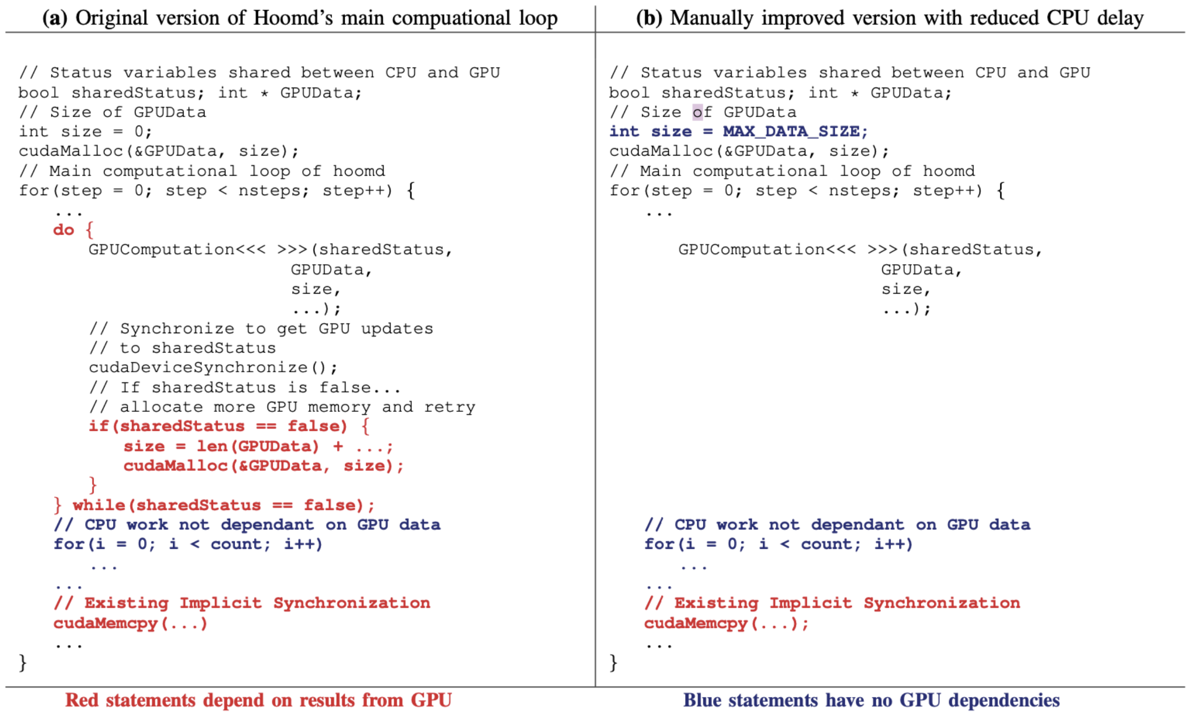
B: *cuFFT library code for function fftw_execute_dft*

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

Synchronizations

Explicit

- Explicit synchronizations are manually placed by the application to synchronize the CPU with the GPU
- In Hoomd-blue the removal of an explicit synchronization operation reduced execution time by 37%
- The explicit sync, shown in to the right in Figure 3, is used to wait for the GPU to update the shared variable sharedStatus which indicates whether the GPU computation failed because not enough memory was allocated
- 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 through N of the for-loop, the application still syncs with the GPU on every iteration causing 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(...)
    ...
}
```

(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(...);
    ...
}
```

Red statements depend on results from GPU

Blue statements have no GPU dependencies

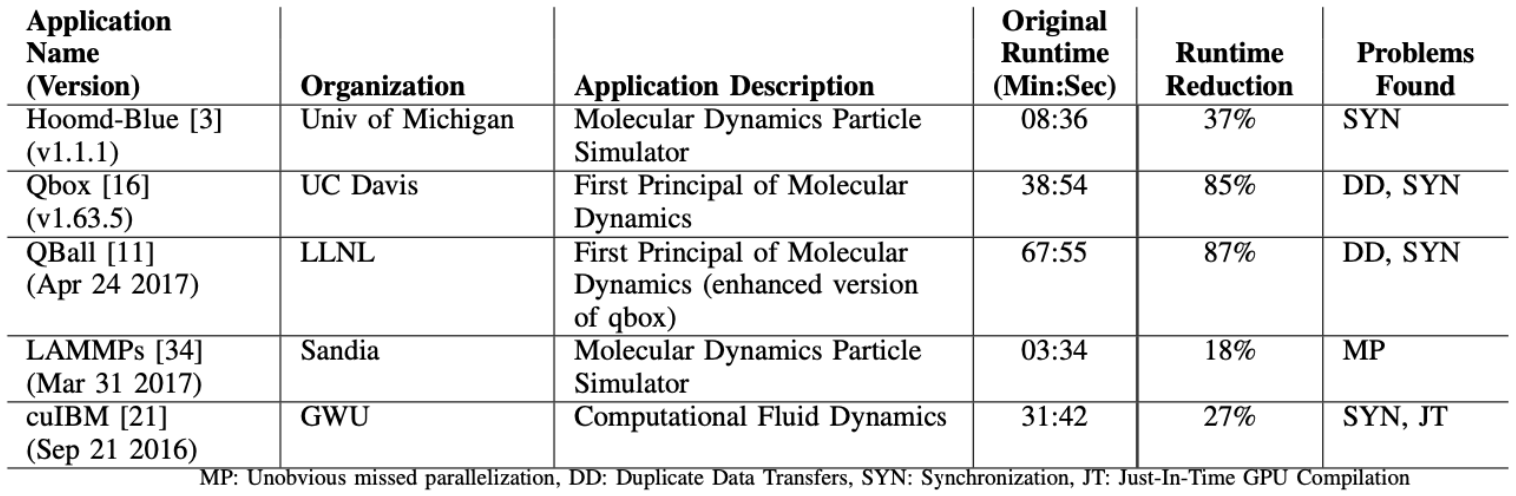
Fig. 3: A flat representation of an explicit synchronization error in the main computational loop in Hoomd.

Synchronizations
Enhancement

- 3 types of unnecessary delay:
  - When the CPU does not access shared data (data shared with the GPU) after the synchronization
  - When the placement of the synchronization is far from the first access of shared data by the CPU
  - When CPU computation not dependent on GPU data is delayed by a synchronization
- The technique to identify unnecessary synchronizations uses profiling, memory tracing and program slicing to identify where a sync opportunity is located and how to correct it
- The approach can be broken down into five steps:
  1. Identify the sync operations that cause long delays on the CPU
  2. Determine what data is shared between CPU and GPU
  3. Identify the CPU instructions that access shared data
  4. Determine how far the instruction performing the sync is from the first CPU instruction that accesses data shared
  5. Determine if CPU computation exists that does not depend on shared data.
- Using this information, the corrective measure and an estimate of the amount of time that could be saved is found

Overall Conclusions

- What links the 3 examined performance issues together is the lack of performance tools and techniques to detect their presence
- The optimization techniques described use a combination of memory tracing, program slicing, content-based data deduplication, and CPU profiling to detect their presence
- Exploiting parallelization of the high-performance scientific applications below resulted in reduction of execution time by 18%-87%!



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

Opinion

I thought the paper was interesting since it connected GPU performance optimization topics covered in SC 5260 and is connected to my research.

I do research in molecular dynamic simulations, so it was cool to see issues with programs I use (Hoomd-blue and LAAMPs). Evaluating how effective the programs really are and optimizing the job submissions on the applications will be very helpful with research.

GPU performance optimization doesn't have to be the application developer's problem! Programmers can make conscious additions that will reduce execution time. The paper gave lots of ideas on where to look for performance modifications.

Paper Reference

El-Araby, E. (2018). Exposing Hidden Performance Opportunities in High Performance GPU Applications. 2018 18TH IEEE/ACM INTERNATIONAL SYMPOSIUM ON CLUSTER, CLOUD AND GRID COMPUTING (CCGRID), 301–310. https://doi.org/10.1109/CCGRID.2018.00045