Improving GPU Performance Prediction with Data Transfer Modeling

Michael Boyer
Department of Computer Science
University of Virginia
mwb7w@cs.virginia.edu

Jiayuan Meng and Kalyan Kumaran
Leadership Computing Facility
Argonne National Laboratory
{jmeng, kumaran}@alcf.anl.gov

***Abstract*—Accelerators such as graphics processors (GPUs) have become increasingly popular for high performance scientific computing. Often, much effort is invested in creating and optimizing GPU code without any guaranteed performance benefit. To reduce this risk, performance models can be used to project a kernel's GPU performance potential before it is ported. However, raw GPU execution time is not the only consideration. The overhead of transferring data between the CPU and the GPU is also an important factor; for some applications, this overhead may even erase the performance benefits of GPU acceleration.**

To address this challenge, we propose a GPU performance modeling framework that predicts both kernel execution time and data transfer time. Our extensions to an existing GPU performance model include a data usage analyzer for a sequence of GPU kernels, to determine the amount of data that needs to be transferred, and a performance model of the PCIe bus, to determine how long the data transfer will take. We have tested our framework using a set of applications running on a production machine at Argonne National Laboratory. On average, our model predicts the data transfer overhead with an error of only 8%, and the inclusion of data transfer time reduces the error in the predicted GPU speedup from 255% to 9%.

I. Introduction

In recent years, graphics processing units (GPUs) have been increasingly used to accelerate scientific computing applications. The motivation is clear: a high-end GPU can provide approximately 6x higher memory bandwidth and 21x higher computational throughput than a high-end CPU.[1] For many applications, these massive computational resources can provide a substantial performance improvement. But the GPU is not a panacea; some applications are not a good fit for the GPU's highly parallel architecture and will see little or no benefit from being ported to a GPU [14]. This realization may not occur, however, until substantial porting effort has already been expended.

To address this challenge, previous work has described GROPHECY, a framework for projecting the best achievable GPU execution time given a high-level code representation referred to as a code skeleton [15]. GROPHECY automatically explores a number of different optimization approaches and projects the execution time for each transformation, without the need to implement and tune GPU code. It takes into account several application-specific factors automatically, such as data parallelism, memory access patterns, control flow divergence, and computational intensity.

However, the GPU computation time alone does not represent the total execution time of a GPU application. A discrete GPU has a physically separate memory from the CPU[2], and data consumed and produced by a GPU kernel must be explicitly copied[3] by the CPU program to and from the GPU's memory across the PCIe bus. For some applications, the time required for this data transfer may be larger than the actual computation time on the GPU; this is true for all of the applications and all but one of the data sets we study in this paper.

Clearly, data transfer overhead is an important factor to consider when determining whether it will be worthwhile to accelerate a workload on a GPU; even if the GPU provides a substantial speedup on the computation, the data transfer overhead may lead to the GPU taking more time overall [4]. Moreover, the amount of data to be transferred depends on the input data and the dataflow among multiple kernels. In some cases, the data transfer overhead is so high that it can only be mitigated if the same data is reused by multiple kernels.

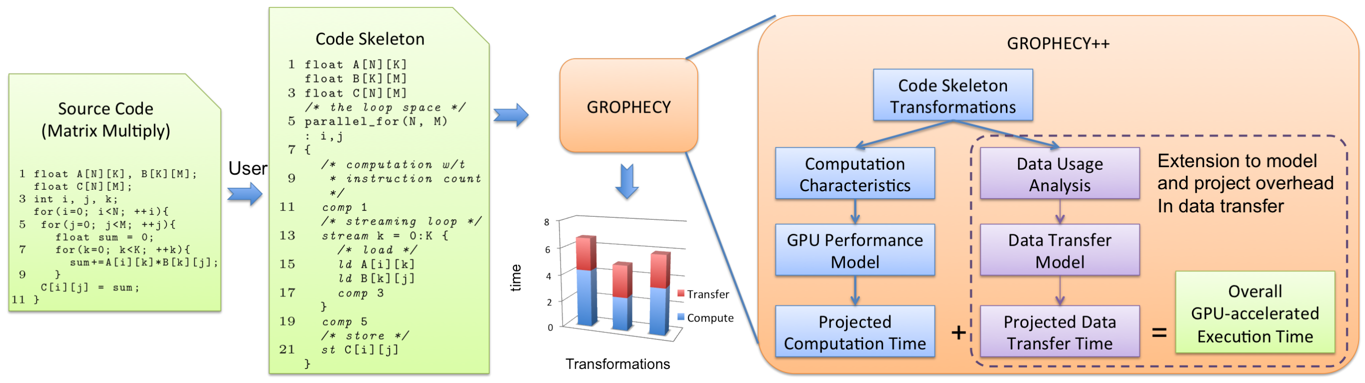
In this paper, we extend GROPHECY to account for data transfer time. The resulting framework, GROPHECY++, provides users with a much more accurate projection of the expected benefit of porting an application to the GPU. Our contributions are three fold:

1) We present a simple but accurate model for predicting PCIe transfer time that requires only two measurements to derive parameters for a specific hardware system.
2) We develop data usage analysis based upon the data flow of the modeled GPU kernels to determine what data needs to be transferred between the CPU and GPU.
3) By integrating the data usage analysis and the transfer time model into GROPHECY and predicting the perfor-

[1]This comparison is based on an AMD Radeon HD 7970 GHz Edition GPU and an Intel Xeon E5-2687W CPU, which provide theoretical peak memory bandwidths of 288 and 51 GB/s, respectively, and computational throughputs of 4,096 and 198 GFLOPS, respectively.

[2]An integrated GPU, on the other hand, physically shares a memory with the CPU. However, integrated GPUs are typically designed for low power rather than high performance, and are not useful for most HPC applications.

[3]In most systems, a kernel running on the GPU can directly access a portion of CPU memory and avoid the explicit copy to and from GPU memory. However, for the majority of applications, this will result in significantly worse performance, because the bandwidth across the PCIe bus is typically an order of magnitude lower than the bandwidth to the GPU's memory.


Source Code
(Matrix Multiply)
User
Code Skeleton
GROPHECY
time
Transfer
Compute
Transformations
GROPHECY++
Code Skeleton Transformations
Computation Characteristics
Data Usage Analysis
Extension to model and project overhead In data transfer
GPU Performance Model
Data Transfer Model
Projected Computation Time
+
Projected Data Transfer Time
=
Overall GPU-accelerated Execution Time

```
float A[N][K], B[K][M];
float C[N][M];
int i, j, k;
for(i=0; i<N; ++i){
 for(j=0; j<M; ++j){
   float sum = 0;
   for(k=0; k<K; ++k){
     sum+=A[i][k]*B[k][j];
   }
 C[i][j] = sum;
}
```

```
float A[N][K]
float B[K][M]
float C[N][M]
/* the loop space */
parallel_for(N, M)
: i,j
{
  /* computation w/t
   * instruction count
   */
  comp 1
  /* streaming loop */
  stream k = 0:K {
    /* load */
    ld A[i][k]
    ld B[k][j]
    comp 3
  }
  comp 5
  /* store */
  st C[i][j]
}
```

Fig. 1. The overall framework of GPU performance projection.

mance of real applications, we empirically demonstrate the importance of modeling data transfer overhead for GPU performance prediction.

Our technique is not application or system specific. We validate the technique using CUDA applications, but the technique could also apply to OpenCL. The data usage analysis is general enough to apply to various workloads. The PCIe bus model is constructed automatically for each new system.

We have experimented with GROPHECY++ on a production data analysis and visualization machine at Argonne National Laboratory, using four different workloads with various input scenarios. On average, the errors in the predicted kernel execution time and data transfer time are only 15% and 8%, respectively. By augmenting the predicted kernel execution time with the predicted transfer time, we reduce the average error in the predicted GPU speedup from 255% to 9%.

II. Background

A. Development of GPU-accelerated Code

In order to accelerate an application using GPUs, one first has to identify time-consuming parts of an application and extract them as kernels. These kernels, provided with sufficient data parallelism, must be re-implemented using a GPU programming language such as OpenCL [9] or CUDA [16].

A significant amount of effort is often spent on tuning a GPU kernel implementation. With GPUs able to execute hundreds of parallel threads, memory throughput often becomes the performance bottleneck. As a result, programmers have to carefully manage data allocation and movement in order to improve reuse and better exploit the computational capabilities. This in turn may require modifications in how a kernel is parallelized. In general, there is a large transformation space to be searched during tuning, and many seemingly promising transformations may result in reduced performance.

B. Data Transfer Overhead

A discrete GPU contains its own memory, separate from the memory used by the CPU. Before executing a kernel, space must be allocated in GPU memory and any requisite input data must be explicitly copied from CPU memory to GPU memory. Likewise, after a kernel completes, any output data required by the CPU must be explicitly copied back from GPU memory. Transfers between GPU and CPU memory occur over the PCI Express (PCIe) bus, which has an effective bandwidth of approximately 3, 6, or 12 GB/s for PCIe versions 1, 2, and 3, respectively. In many systems, the PCIe bandwidth may be an order of magnitude slower than the CPU's memory bandwidth and close to two orders of magnitude slower than the GPU's memory bandwidth. The overhead of transferring data between the CPU and the GPU can potentially outweigh any performance improvements due to GPU execution.

For example, consider a simple program that adds together two large vectors. Vector addition is an extremely data parallel operation, which would appear to make it particularly well suited to the GPU. Assume we plan to run the program on a system containing an Intel Xeon E5645 CPU and an NVIDIA Quadro FX 5600 GPU, which provide peak memory bandwidths of 32 and 77 GB/s, respectively. Because vector addition involves little computation on each element, we would expect the program to be bandwidth-bound on both devices. Therefore, we would expect the GPU to complete the vector addition approximately $\frac{77}{32} = 2.4$x faster than the CPU.

But if the input vectors are located in CPU memory, we must first transfer them to GPU memory. Similarly, if the output vector will be consumed by a program running on the CPU, then we must transfer it back to CPU memory. Both of these transfers occur over the PCIe bus, which has an effective bandwidth of about 3 GB/s. Thus, taking into account data transfer time, the CPU will actually complete the entire vector addition about $\frac{32}{3} \approx 10$x faster than the GPU. For many applications, we must consider data transfer time in order to make an appropriate decision about whether to leverage a GPU.

C. GPU Performance Projection Framework

Due to the uncertainty in performance gains and the cost of tuning GPU code, application developers often ponder the viability of using GPUs to benefit their science and whether it is indeed worth investing the time and effort to port their code. However, to obtain a reasonable understanding of the GPU performance potential, one needs to consider the dynamics between the system hardware and the application behavior. In particular, it is necessary to estimate how the application may be transformed to exploit the hardware capability to its maximum. Different transformations may result in performance that is orders of magnitude apart.

We have previously proposed a GPU performance projection framework named GROPHECY [15], which estimates the GPU performance of a computation kernel. Figure 1 illustrates the overall framework using a pedagogical example of matrix multiplication. The input to GROPHECY is a simplified description of the corresponding CPU code, referred to as a *code skeleton*. A code skeleton summarizes the high level semantics of a kernel, including loops, parallelism, computation intensity, and data access patterns. With the code skeleton, GROPHECY is able to explore various code transformations, synthesize performance characteristics for each transformation, and then supply the characteristics to a GPU performance model to project the execution time of a particular transformation. The GPU performance model can be configured to reflect different GPU architectures. Eventually, GROPHECY projects the best achievable performance and the transformations necessary to reach that performance.

GROPHECY, however, only models GPU computation time and does not account for data transfer overhead. In order to project the overall performance resulting from GPU acceleration, we extend GROPHECY to include a performance model for CPU-GPU data transfer and the capability to detect what data needs to be transferred.

III. Projecting Data Transfer Overhead

A. Framework Overview

We have added to GROPHECY the capability to model and project data transfer overhead between the CPU and GPU. The extended framework, GROPHECY++, can project how a piece of CPU code may perform if offloaded to a GPU, without implementing and tuning GPU code.

The dashed box in Figure 1 illustrates where our extension lies in the overall framework. The extension has two major components: a data usage analyzer and an empirical model of the PCIe bus. Based on the transformed code skeleton proposed internally by GROPHECY++, the data usage analyzer determines what data needs to be transferred to and from the GPU. The time for each transfer is calculated using the PCIe bus model. Their sum, the projected data transfer overhead, is then added to the projected kernel execution time to obtain the overall GPU-accelerated performance of the workload. The rest of this section describes these two components in more detail.

B. Identifying Transferred Data

As described in prior work [15], a Bounded Regular Section (BRS) [5] can be used to represent the range of array elements accessed by each statement across all nesting loops. The `INTERSECT` operator detects overlap among BRSs and the `UNION` operator merges BRSs. These operations, combined with information about whether an access is a `load` or a `store`, allow GROPHECY to determine the dependencies among BRSs.

To determine what data needs to be transferred from the CPU to the GPU, we maintain a list of BRSs that are read but are not previously written. The `UNION` of all such BRSs is data that needs to be transferred to the GPU. The `UNION` of all written BRSs is data that needs to be transferred back from the GPU. Users can optionally provide hints to specify written data that serve as temporaries. Temporary data need not be transferred back to the CPU. Each individual array is assumed to be transferred separately, although in practice transferring multiple small arrays together as one may provide a minor performance benefit at the cost of more substantial program modifications.

In irregular applications such as sparse linear algebra, the BRS is unknown. In such scenario, GROPHECY++ uses the conservative assumption that all elements in the sparse array may be referenced, and therefore must be transferred, unless users provide additional hints.

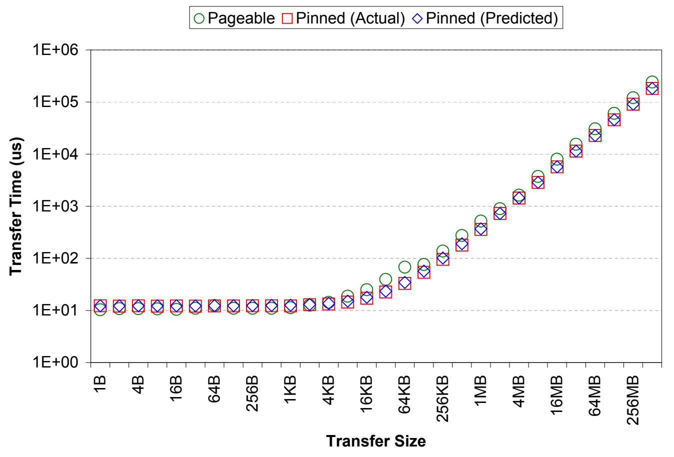
C. Modeling Data Transfer Overhead

One way to potentially reduce the data transfer overhead is to take advantage of *pinned* (page-locked) CPU memory, allocated using the CUDA function `cudaHostAlloc`, which is an alternative to the more common *pageable* CPU memory, allocated using the C standard library function `malloc`. Figure 2 shows the transfer time using pinned and pageable memory for both CPU-to-GPU and GPU-to-CPU transfers. Note that the Y-axis (and X-axis) is log-scaled, which understates the performance difference at large transfer sizes.

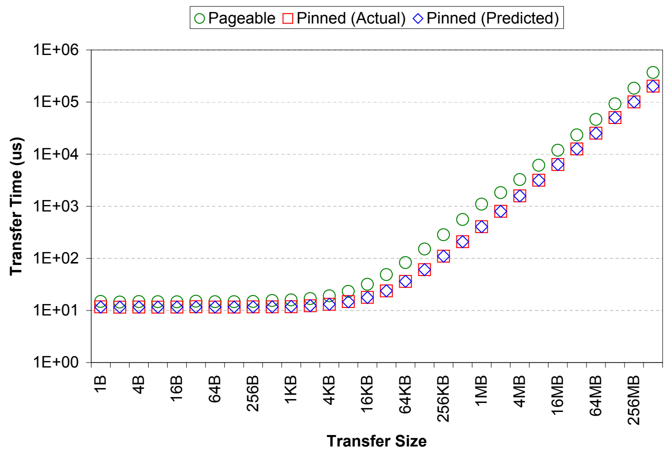
With the exception of CPU-to-GPU transfers smaller than 2KB, a transfer using pinned memory is always faster than an equivalent transfer using pageable memory. Figure 3 demonstrates this clearly by plotting the speedup of pinned memory transfers relative to pageable memory transfers. In this paper, we assume the use of pinned memory since it is advantageous in most typical use cases. In future work we may extend our framework to automatically explore the tradeoff between the two types of memory.

Although we might hope to model the data transfer time in an analytical way, there exists too much variation in transfer times across different systems due to both hardware and software differences. Instead, we construct a simple empirical model, based on only two parameters, that is accurate as well as inexpensive to generate and use. GROPHECY++ automatically measures the values of the two parameters for each new system on which it runs.

We can see from Figure 2 that the time to transfer data across the PCIe bus between the CPU and GPU memories consists of a fixed overhead representing the latency of sending the first byte, α, plus the time required to send each subsequent byte, β, which corresponds to the inverse of the transfer bandwidth. Thus, the time to transfer d bytes can be modeled



(a) CPU to GPU.



(b) GPU to CPU.

Fig. 2. Transfer time for pinned and pageable memory for a range of transfer sizes. Note that both axes are log-scaled. Each time is the arithmetic mean of 10 separate transfers, measured on the system described in Section V.

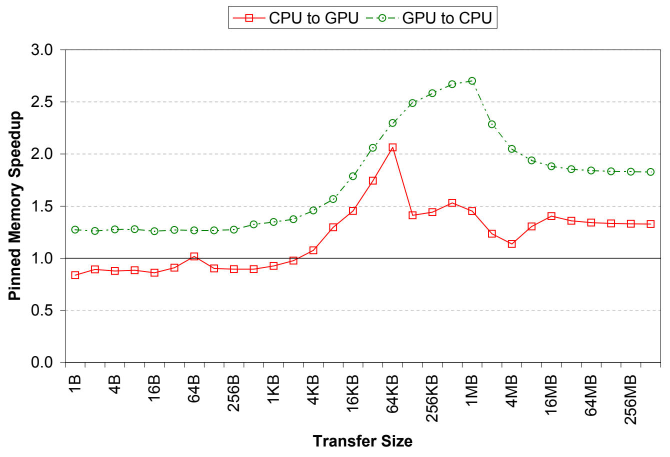


Fig. 3. Speedup of transfers using pinned memory relative to transfers using pageable memory for a range of transfer sizes.

with the following linear[4] equation:

$$T(d) = \alpha + \beta d \qquad (1)$$

For small transfers (<1KB), the α term dominates and the slope is effectively zero; for large transfers (>1MB), the βd term dominates. On the system we use in this paper, α is on the order of 10 us and the transfer bandwidth ($1/\beta$) is approximately 2.5 GB/s.

To determine α, we measure the transfer time t_S of a single byte; we then set $\alpha = t_S$. To determine β, we measure the time t_L of a large transfer of size $s_L = 512\text{MB}$[5] and then set $\beta = \frac{t_L}{s_L}$. Both t_S and t_L are averaged across ten runs to reduce the impact of noise. These two measurements are performed by a simple synthetic benchmark, which is automatically invoked by GROPHECY++ when run on a new system. GROPHECY++ then uses the linear model given in Equation 1 to predict expected transfer times.

To demonstrate the accuracy of the simple linear model, Figure 2 plots both the predicted transfer time and the measured transfer time for transfers using pinned memory. For both directions of transfer, the predicted time closely matches the actual time. More detailed validation of the model is presented in Section V-A.

[4]Transfers using pageable memory exhibit slightly more non-linear behavior at intermediate transfer sizes.

[5]512MB is chosen rather arbitrarily; any size larger than a few megabytes would be sufficient. Choosing a data size close to the maximum size supported by the system is a reasonable approach in general.

IV. Methodology

A. Experimental Setup

As described previously, GROPHECY explores a number of different optimization strategies for a given kernel and predicts the resulting performance for each [15]. For the purposes of this paper, the predicted kernel execution time is the predicted time of the best-performing version of the kernel. The real kernel execution time is measured using a hand-coded version of the kernel that employs the same optimization strategies

TABLE I

MEASURED KERNEL EXECUTION AND DATA TRANSFER TIMES AND DATA TRANSFER SIZES FOR EACH APPLICATION AND DATA SIZE. THE "PERCENT TRANSFER" COLUMN SHOWS THE FRACTION OF THE OVERALL TIME THAT IS DUE TO DATA TRANSFER.

		Time (ms)		**Percent**	**Total Transfer Size (MB)**	
Application	**Data Size**	**Kernel**	**Transfer**	**Transfer**	**Input**	**Output**
CFD	97K	1.9	3.2	63	6.3	1.9
	193K	3.2	6.2	66	12.6	3.7
	233K	3.1	7.4	70	15.1	4.4
HotSpot	64 x 64	< 0.1	< 0.1	41	< 0.1	< 0.1
	512 x 512	0.3	1.2	77	2.0	1.0
	1024 x 1024	1.2	4.6	79	8.0	4.0
SRAD	1024 x 1024	2.0	4.0	67	4.0	4.0
	2048 x 2048	7.6	13.0	63	16.0	16.0
	4096 x 4096	28.1	49.0	64	64.0	64.0
Stassuij		2.4	4.9	67	8.5	4.1

suggested by GROPHECY. The predicted data transfer time is generated using the linear model described earlier. The real data transfer time is measured using the same manually ported CUDA implementation, which employs pinned memory.

The total GPU time, whether predicted or measured, is simply the sum of the kernel and transfer times. For applications with multiple kernels, the total GPU time is the sum of the execution times of all kernels and their collective data transfer times. The total CPU time is the execution time of the same portion of the application that has been ported to the GPU. The GPU speedup is the total CPU time divided by the total GPU time. For all experiments, the measured time represents the arithmetic mean of ten separate runs. For iterative applications, the number of iterations is one unless otherwise noted.

We run the CPU and GPU implementations on a single node of a production data analysis and visualization cluster at Argonne National Laboratory. The system contains a hyper-threaded quad-core Intel Xeon E5405 CPU running at 2.00 GHz and an NVIDIA Quadro FX 5600 GPU; the GPU is a PCIe v1 device and is installed in an x16 slot. The system is configured with SUSE Linux Enterprise Server 10 SP3, CUDA 2.3, and NVIDIA graphics driver version 260.19.36.

B. Benchmarks

To quantify the impact of augmenting GROPHECY with a data transfer model, we use four benchmarks that are key components in representative applications in the areas of medical imaging, microprocessor design, fluid dynamics, and quantum physics. `SRAD`, `HotSpot`, and `CFD` are benchmarks found in the Rodinia benchmark suite [2]. `Stassuij` is extracted from a production application in DOE's INCITE program. The three Rodinia applications all have multiple data sets of varying sizes.

The baseline implementations of the benchmarks are implemented in C++. The GPU implementations are parallelized using CUDA, while the CPU implementations are parallelized using OpenMP and executed with 8 threads.

`CFD` is an unstructured-grid, finite-volume solver for the 3D Euler equations for compressible flow. The core part of the benchmark is spread over three GPU kernels. The two kernels are separated in order to enforce global synchronization so that an array can be consumed before it is updated. The data size in CFD represents the number of particles being simulated.

`HotSpot` is an ordinary differential equation solver over a structured grid which is used to estimate micro-architecture temperature. Every element is computed by gathering a 3×3 neighborhood of elements (i.e., the stencil) from the input array. Multiple invocations of the same kernel across several iterations can be fused together. The data size in HotSpot represents the size of the grid.

`SRAD` is a diffusion method to remove speckles from ultrasonic and radar imaging applications without destroying important image features. It has two kernels: the first one generates diffusion coefficients, and the second one updates the image. Data dependency among the two kernels involves several arrays, and each data-parallel task in the consumer kernel depends on several tasks in the producer kernel. The data size in SRAD represents the size of the image.

`Stassuij` lies in the core of Green's Function Monte Carlo [8], [18], which performs Monte Carlo calculations for light nuclei. It multiplies a 132×132 sparse matrix of real numbers with a 132×2048 dense matrix of complex numbers. The sparse matrix is represented in CSR format with three vectors.

CFD, HotSpot, and SRAD are all iterative applications. In CFD, three separate kernels are invoked each iteration; in HotSpot and SRAD, each iteration corresponds to a single kernel invocation. For all three applications, the amount of data transferred is independent of the number of iterations: a fixed amount of input data is transferred to the GPU before the first iteration, and a fixed amount of output data is transferred back to the CPU after the final iteration. The relative performance of the GPU and CPU depends on the number of iterations: as the number of iterations grows, the data transfer overhead is amortized over a larger amount of computation, and the speedup of the GPU over the CPU increases. If we ignore the data transfer time, the speedup is fixed regardless of the iteration count.

Table I shows the measured kernel execution time and data transfer time for all four applications, as well as the fraction of overall time consumed by data transfer and the total amount of data transferred. For all applications and data sets, with the exception of HotSpot's smallest data set, the transfer time is greater than the kernel execution time. This means that

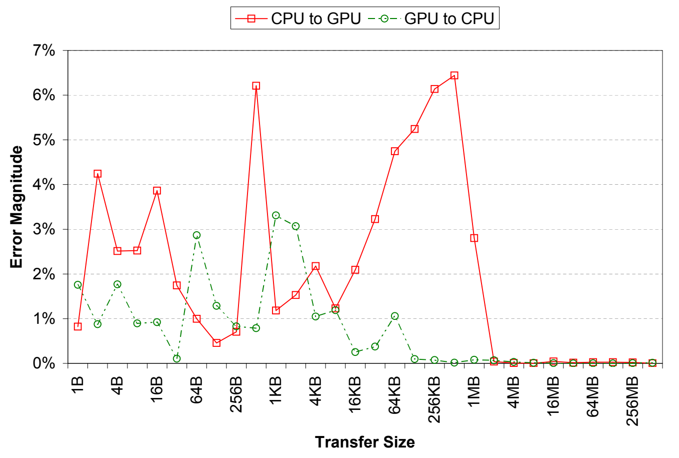


Fig. 4. Absolute value of the percent difference between the predicted and measured transfer times for transfers to and from the GPU across a range of transfer sizes.

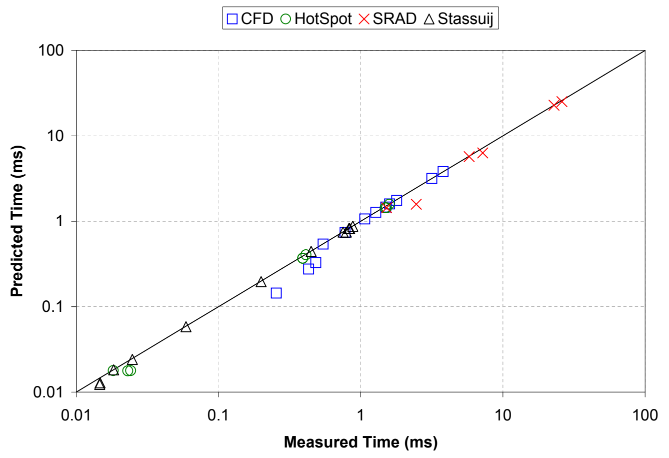


Fig. 5. Predicted transfer time versus measured transfer time for each individual transfer across all applications and data sizes. A perfect prediction would fall on the plotted $y = x$ line. Transfers that are slower than predicted fall below the line.

accurately predicting transfer times is crucial to providing a useful estimate of the overall GPU execution time.

V. Results

We first validate the proposed data transfer model using a synthetic benchmark, and then explore the impact of using the model to predict the overall performance of real applications.

A. Data Transfer Model Validation

To validate the transfer time model, we use a synthetic benchmark to measure the actual transfer time for all transfer sizes that are a power of two, ranging from 1B to 512MB. To characterize the model's accuracy, we use the *error magnitude*, defined as the absolute value of the percent difference between the predicted and measured values. Figure 4 shows the error magnitude for each transfer size, for transfers both to and from the GPU. The maximum errors are 6.4% and 3.3% for CPU-to-GPU and GPU-to-CPU transfers, respectively. The relative error is larger at smaller data sizes and is essentially zero for all transfer sizes larger than 1MB.

To quantify the overall prediction error, we compute the arithmetic mean error magnitude across all transfer sizes. The average errors are only 2.0% and 0.8%, respectively. Almost all of error in the GPU-to-CPU prediction and approximately half of the error in the CPU-to-GPU prediction is due to inherent variation in the transfer time. If we run the synthetic benchmark across the complete range of transfer sizes twice, and use the measured values from the first run to predict the values we expect to observe in the second run, the average errors are 1.0% and 0.7%, respectively.

Figure 5 plots the predicted versus measured transfer time of every transfer for all applications and data sizes. Most predictions are quite accurate, with a few notable outliers. The smallest transfers in HotSpot (the circles in the bottom left corner of the chart) are of the same size and therefore have the same predicted transfer time, but have significantly different measured times. The three slow transfers in CFD (the squares near the center of the chart) represent a particular transfer that, inexplicably, has high variability. In approximately half of the runs, the measured time matches the predicted time fairly closely, but in the other half of the runs, the measured time is more than two times slower than the predicted time. Overall, the average prediction error across all transfers is 7.6%.

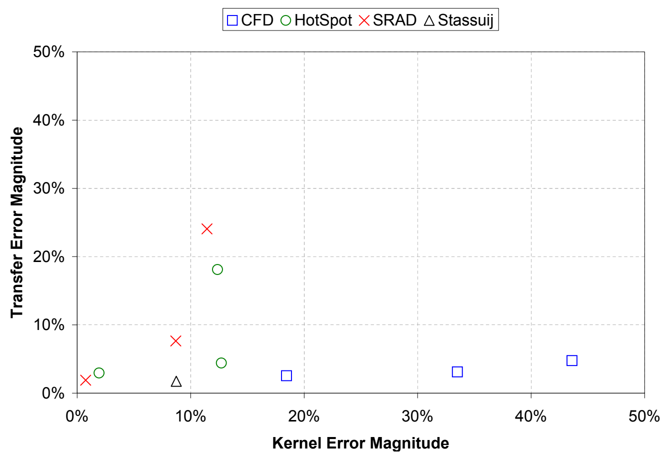


Fig. 6. Error magnitude of transfer predictions versus error magnitude of kernel predictions. The transfer error is the overall error across all of the transfers for a single data size of a given application. Likewise, for applications with multiple kernels, the kernel error is the overall error across all of the kernels.

B. Application Performance

Figure 6 plots the overall transfer prediction error magnitude versus the overall kernel prediction error magnitude for all applications and data sizes. For CFD, the kernel prediction error dominates. For HotSpot and SRAD, both kernel and transfer errors are relatively small at most data sizes, on the order of 10% or less. The transfer error is roughly twice the kernel error at the smallest data sizes for both applications. Note that we present the error *magnitude* (the absolute value

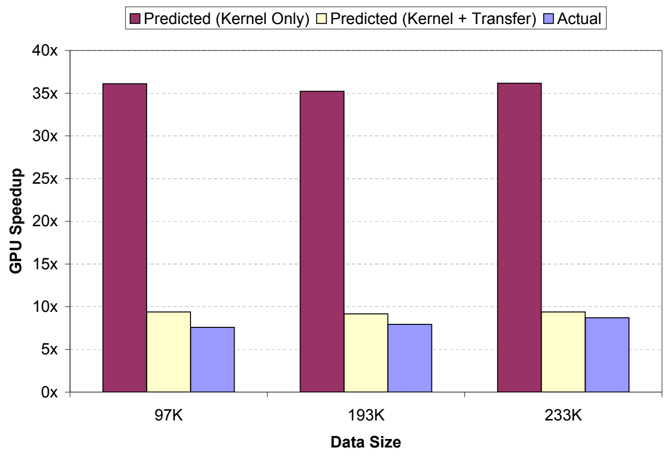


Fig. 7. Measured and predicted GPU speedup for CFD across a range of data sizes. Predictions both with and without data transfer time are shown.

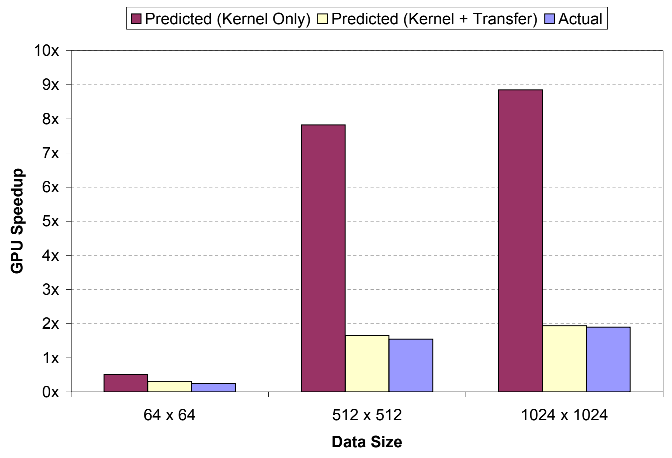


Fig. 9. Measured and predicted GPU speedup for HotSpot across a range of data sizes. Predictions both with and without data transfer time are shown.

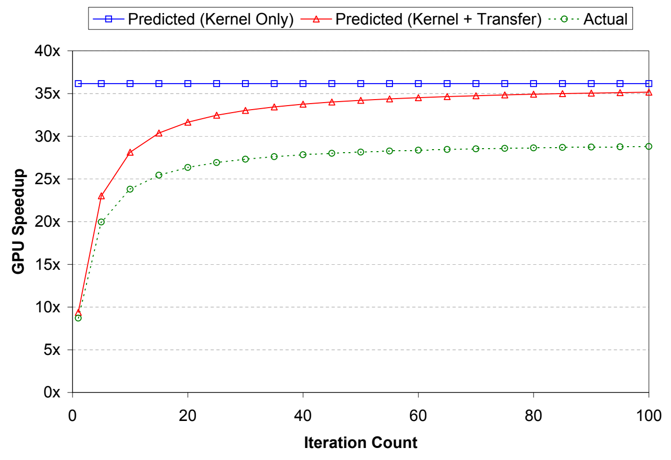


Fig. 8. Measured and predicted GPU speedup of CFD as a function of iteration count for a data size of 233K. Predictions both with and without data transfer time are shown.

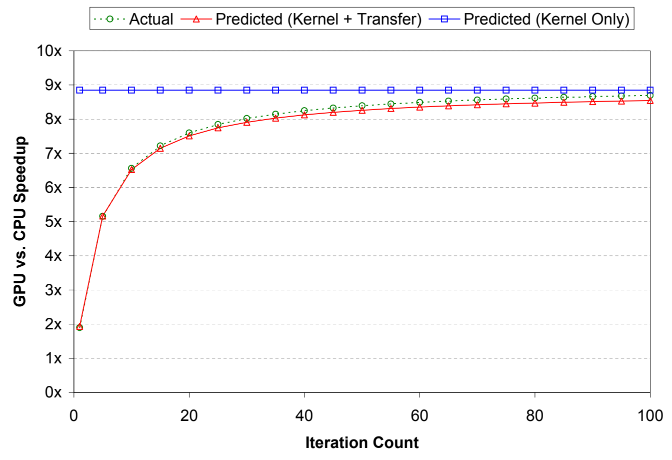


Fig. 10. Measured and predicted GPU speedup of HotSpot as a function of iteration count for a data size of 1024 x 1024. Predictions both with and without data transfer time are shown.

of the error) for simplicity. But all of the errors, with the exception of one of the kernel predictions for HotSpot and the only kernel prediction for Stassuij, are actually negative, meaning that the predicted time is *less* than the measured time.

1) CFD: Figure 7 shows the predicted and measured speedup of CFD for three different data sizes. Accounting for the transfer time in CFD is critical because the transfer time makes up, on average, two thirds of the total execution time. Because of this, and also because the kernel time is underpredicted by an average of 32%, the predicted GPU speedup that only takes into account kernel time is more than four times the actual speedup! By accounting for the overhead of data transfer, we can predict the GPU speedup with an error of only 16%.

Figure 8 shows the predicted and measured speedup for CFD's largest data size as the number of iterations increases. Taking into account the data transfer time greatly increases the prediction accuracy for small iteration counts; the predicted speedup with data transfer time remains more than twice as accurate (i.e., the error magnitude is more than twice as small) for iteration counts less than 18. As the iteration count grows large, the transfer overhead becomes negligible relative to the execution time, and the predicted speedups both with and without the transfer time converge. In the limit, as the number of iterations approaches infinity, the error is 22.6%.

2) HotSpot: Figure 9 shows the predicted and measured speedup of HotSpot for three different data sizes. At the smallest data size, despite a relatively large 18% error in the predicted transfer time, the predicted overall GPU speedup has an error of only 17%. At larger data sizes, the prediction error is only 4%. However, without modeling data transfer, the predicted GPU speedup is two times higher than reality for small data sizes and four times higher for large data sizes.

Figure 10 shows the measured and predicted speedup as a function of the number of iterations for HotSpot's largest data size. Unlike for CFD, the predicted speedup that takes into

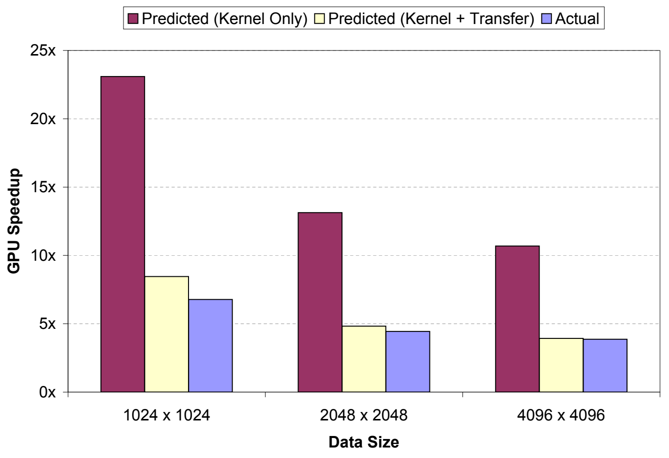


Fig. 11. Measured and predicted GPU speedup for SRAD across a range of data sizes. Predictions both with and without data transfer time are shown.

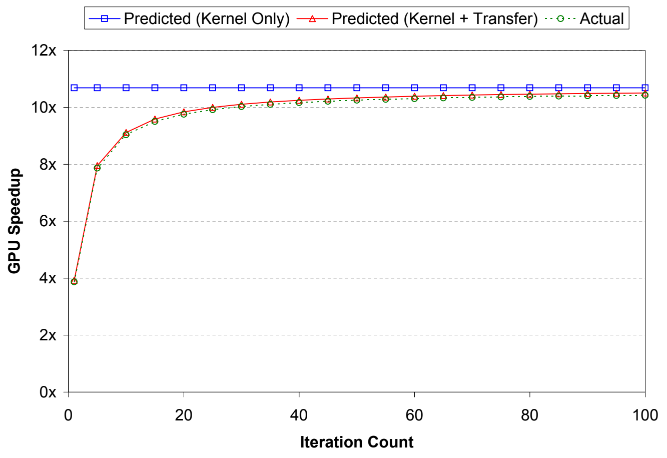


Fig. 12. Measured and predicted GPU speedup of SRAD as a function of iteration count for a data size of 4096 x 4096. Predictions both with and without data transfer time are shown.

account data transfer time remains fairly accurate even for large iteration counts, due to a better prediction of the kernel execution time. The predicted speedup with data transfer time remains more than twice as accurate as the predicted speedup without transfer time for iteration counts as large as 70. In the limit, both prediction methods have an error of only 1.9%.

3) SRAD: Figure 11 shows the predicted and measured speedup of SRAD for three data sizes. The error in the predicted transfer time at the smallest data size is 24%, larger than for any other data size in any application. Despite this relatively large error, accounting for data transfer time still significantly increases the overall prediction accuracy, reducing the error in predicted GPU speedup by nearly tenfold, from 241% to 25%. At the larger data sizes, the improvement is even greater: the average speedup error is reduced from 186% to 5%.

Figure 12 shows the measured and predicted speedup for different iteration counts for SRAD's largest data size. The data-transfer-aware prediction is extremely accurate at all iteration counts, because the errors in both the predicted kernel time (0.7%) and the predicted transfer time (1.9%) are so small. For all iteration counts less than 228, the data-transfer-aware prediction is more than twice as accurate as the kernel-only prediction. In the limit, as the number of iterations goes to infinity, the prediction error is only 0.75%.

4) Stassuij: For the previous three applications, not taking into account data transfer time resulted in a significantly overpredicted GPU speedup, but it did *not* result in a misprediction of whether the GPU implementation would be faster or slower than the CPU implementation.[6] In other words, despite significant mispredictions in the *magnitude* of the speedup, the kernel-only prediction still correctly predicted *whether or not the speedup would be greater than one*[7]. Stassuij is unlike the previous applications in that regard: the kernel-only predicted speedup is 1.10x, indicating an overall increase in performance from using the GPU, but the actual GPU speedup, with data transfer overhead included, is only 0.39x, representing an overall slowdown. Taking into account data transfer time results in a predicted speedup of 0.38x, an error of only 1.6%.

C. *Overall Performance*

Table II shows the error in the predicted speedup for all applications and data sets. In addition to the error achieved by predictions based on only the kernel time and both the kernel time and the data transfer time, we also present the error achieved by predictions based on only the data transfer time. Because data transfer time represents around two-thirds of the total execution time for most of the kernels we study, using the transfer time instead of the kernel execution time to predict the overall GPU speedup reduces the average prediction error across the four applications from 255% to 68%. Using both the transfer and kernel times together further reduces the average prediction error to only 9%.

VI. Related Work

Performance models have been increasingly used for application tuning over complex or large scale systems [3], [12], [17]. These models target performance over a cluster or a heterogeneous platform, with a focus on the modeling and optimization of communication and scheduling among nodes. Lee et al. [13] used machine learning techniques to model the relationship between tunable parameters and application

[6]This is not to discount the importance of the magnitude of the predicted speedup. For example, the speedup of 7.8x predicted (using only the kernel time) for one of the data sets for HotSpot makes a compelling argument for porting that kernel to the GPU, and perhaps might even justify the purchase of special hardware; the actual speedup of 1.5x is much less compelling.

[7]We mention a speedup cutoff of one here because it is intuitive, but such a cutoff might be too low in practice. Even if a programmer were able to achieve a speedup slightly greater than one by using the GPU, the small performance gain might not be worth the significant effort, or perhaps an even greater gain would have been achieved by devoting that same effort towards optimizing the CPU implementation.

TABLE II
ERROR MAGNITUDE OF THE PREDICTED GPU SPEEDUP USING ONLY THE PREDICTED KERNEL EXECUTION TIME, ONLY THE PREDICTED DATA TRANSFER TIME, OR THE COMBINATION OF BOTH PREDICTED TIMES. TWO OVERALL AVERAGES ARE PRESENTED: ONE THAT WEIGHTS ALL DATA SETS EQUALLY (AND THUS WEIGHTS MULTI-DATA-SET APPLICATIONS MORE HEAVILY) AND ONE THAT WEIGHTS ALL APPLICATIONS EQUALLY.

		Error in Predicted GPU speedup		
Application	**Data Set**	**Kernel Only**	**Transfer Only**	**Kernel and Transfer**
CFD	97K	377%	67%	24%
	193K	344%	56%	15%
	233K	316%	46%	8%
	Average	345%	56%	16%
HotSpot	64x64	93%	198%	17%
	512x512	406%	35%	7%
	1024x1024	366%	31%	2%
	Average	288%	88%	9%
SRAD	1024x1024	241%	97%	25%
	2048x2048	196%	72%	9%
	4096x4096	176%	61%	1%
	Average	204%	76%	12%
Stassuij		182%	51%	2%
Average (data sets)		270%	71%	11%
Average (applications)		255%	68%	9%

performance. Snavely et al. [19] proposed a framework for performance modeling and prediction, in which traces are used to characterize application signatures. All of these techniques must measure or profile application statistics over each type of node and assume code already exists for each architecture.

Several GPU performance models have been proposed [1], [6], [10], [21]. These techniques predict and analyze the performance of GPU codes for accessible GPUs. However, GPU performance models alone are not able to transform CPU code structures for optimization on GPUs, let alone project GPU performance from CPU code. In addition, they do not take into consideration the overhead of data transfer between the CPU and the GPU. We provide a holistic approach to project GPU performance from CPU code, by modeling both code transformations and data transfer overhead.

There exist compiler techniques that automate the data transfer between the CPU and GPU [7]. By analyzing data usage at the granularity of an allocation unit, the data consumed and produced by the GPU kernel can be identified implicitly. Our performance modeling framework could help such a technique optimize the compiler transformation, by identifying which array sections need to be transferred and whether it is worthwhile to offload data to the GPU.

Several other empirical techniques have been proposed to project cross-platform performance. Yang et al. [20] profiled partial execution of an application on different platforms to infer relative full-application performance. Lee and Brooks [11] estimated application performance over different microarchitectural configurations using regression modeling, which was trained according to an initial set of performance data. Different from the above approaches, our projection is cast without the need to re-implement the code on a different platform. In addition, we also project the data movement overhead for partially offloading a workload to a different architecture.

VII. CONCLUSION

In this paper, we present GROPHECY++, a framework that projects the overall GPU speedup from abstract CPU code. We extend the existing GROPHECY framework with the capability to model and project the overhead of data transfer between the CPU and GPU. The key components in our extension include a data usage analyzer and a PCIe bus model. The former determines the amount of data to be transferred given a sequence of kernels, and the latter estimates the transfer time given the amount of data. By including the data transfer overhead in the predicted performance, users are provided with a more holistic estimation of the performance that can be gained from GPU acceleration and whether it is worthwhile to port their existing CPU code to the GPU.

We demonstrate in this paper that data transfer overhead is an important factor to consider when modeling performance. The overall speedup with and without data transfer can differ by 7x or more. In fact, for some workloads, the data transfer overhead may turn a GPU speedup into a slowdown. We have tested GROPHECY++ with four benchmarks, utilizing different input data sets and varying numbers of iterations. On average, we achieve a prediction error of 8% on the data transfer overhead and 9% on the overall GPU speedup.

For future work, we plan to expand the scope of the data transfer overhead modeling to explore the tradeoffs of using different types of memory (i.e., pinned and pageable) and account for the overhead of memory allocation. In addition, we plan to validate our model on a wider range of applications as well as hardware systems.

ACKNOWLEDGMENT

We thank the ALCF application and operations support staff for their help. We also thank Venkatram Vishwanath and Vitali Morozov for their helpful discussions. We gratefully acknowledge the help provided by the application teams whose codes are used herein.

This work was supported in part by a GRC AMD/Mahboob Khan Ph.D. fellowship in association with GRC task 1972.001. This research used resources of the Argonne Leadership Computing Facility at Argonne National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under contract DE-AC02-06CH11357.

References

[1] S. S. Baghsorkhi, M. Delahaye, S. J. Patel, W. D. Gropp, and W.-m. Hwu. An adaptive performance modeling tool for GPU architectures. In *PPoPP*, 2010.

[2] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *International Symposium on Workload Characterization (IISWC)*, October 2009.

[3] H. Gahvari, A. H. Baker, M. Schulz, U. M. Yang, K. E. Jordan, and W. Gropp. Modeling the performance of an algebraic multigrid cycle on HPC platforms. In *ICS*, 2011.

[4] Chris Gregg and Kim Hazelwood. Where is the data? Why you cannot debate CPU vs. GPU performance without the answer. In *International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2011.

[5] P. Havlak and K. Kennedy. An implementation of interprocedural bounded regular section analysis. *IEEE Trans. Parallel Distrib. Syst.*, 2, 1991.

[6] S. Hong and H. Kim. An analytical model for a GPU architecture with memory-level and thread-level parallelism awareness. In *ISCA*, 2009.

[7] T. B. Jablin, P. Prabhu, J. A. Jablin, N. P. Johnson, S. R. Beard, and D. I. August. Automatic CPU-GPU communication management and optimization. In *PLDI*, 2011.

[8] M. H. Kalos, M. A. Lee, P. A. Whitlock, and G. V. Chester. Modern potentials and the properties of condensed ^{4}He. In *Phys. Rev. C 66, 044310-1:14*, 1981.

[9] Khronos Group Std. The OpenCL Specification, Version 1.0. http://www.khronos.org/registry/cl/specs/opencl-1.0.33.pdf, 2009.

[10] K. Kothapalli, R. Mukherjee, M. S. Rehman, S. Patidar, P. J. Narayanan, and K. Srinathan. A performance prediction model for the CUDA GPGPU platform. In *HiPC*, 2009.

[11] B. C. Lee and D. M. Brooks. Accurate and efficient regression modeling for microarchitectural performance and power prediction. In *ASPLOS-XII*, 2006.

[12] B. C. Lee, J. Collins, H. Wang, and D. Brooks. CPR: Composable performance regression for scalable multiprocessor models. In *MICRO*, 2008.

[13] Benjamin C. Lee, David M. Brooks, Bronis R. de Supinski, Martin Schulz, Karan Singh, and Sally A. McKee. Methods of inference and learning for performance modeling of parallel applications. In *PPoPP*, 2007.

[14] V. W. Lee, C. Kim, J. Chhugani, M. Deisher, D. Kim, A. D. Nguyen, N. Satish, M. Smelyanskiy, S. Chennupaty, P. Hammarlund, R. Singhal, and P. Dubey. Debunking the 100x GPU vs. CPU myth: an evaluation of throughput computing on CPU and GPU. In *ISCA*, 2010.

[15] Jiayuan Meng, Vitali A. Morozov, Kalyan Kumaran, Venkatram Vishwanath, and Thomas D. Uram. GROPHECY: GPU performance projection from CPU code skeletons. In *SC*, 2011.

[16] NVIDIA Corporation. NVIDIA CUDA compute unified device architecture programming guide. http://developer.download.nvidia.com/compute/cuda/ 08/NVIDIA_CUDA_Programming_Guide_0.8.pdf, 2007.

[17] J. A. Pienaar, A. Raghunathan, and S. Chakradhar. MDR: performance model driven runtime for heterogeneous parallel platforms. In *ICS*, 2011.

[18] S. C. Pieper, K. Varga, and R. B. Wiringa. Quantum Monte Carlo calculations of A=9,10 nuclei. In *Phys. Rev. C 66, 044310-1:14*, 2002.

[19] A. Snavely, L. Carrington, N. Wolter, J. Labarta, R. Badia, and A. Purkayastha. A framework for performance modeling and prediction. In *SC*, 2002.

[20] L. T. Yang, X. Ma, and F. Mueller. Cross-platform performance prediction of parallel applications using partial execution. In *SC*, 2005.

[21] Y. Zhang and J. D. Owens. A quantitative performance analysis model for GPU architectures. In *HPCA*, 2011.