

Where is the Data? Why You Cannot Debate CPU vs. GPU Performance Without the Answer

Chris Gregg
Department of Computer Science
University of Virginia

Kim Hazelwood
Department of Computer Science
University of Virginia

Abstract—General purpose GPU Computing (GPGPU) has taken off in the past few years, with great promises for increased desktop processing power due to the large number of fast computing cores on high-end graphics cards. Many publications have demonstrated phenomenal performance and have reported speedups as much as 1000x over code running on multi-core CPUs. Other studies have claimed that well-tuned CPU code reduces the performance gap significantly. We demonstrate that this important discussion is missing a key aspect, specifically the question of where in the system data resides, and the overhead to move the data to where it will be used, and back again if necessary. We have benchmarked a broad set of GPU kernels on a number of platforms with different GPUs and our results show that when memory transfer times are included, it can easily take between 2 to 50x longer to run a kernel than the GPU processing time alone. Therefore, it is necessary to either include memory transfer overhead when reporting GPU performance, or to explain why this is not relevant for the application in question. We suggest a taxonomy for future CPU/GPU comparisons, and we argue that this is not only germane for reporting performance, but is important to heterogeneous scheduling research in general.

I. INTRODUCTION

General-purpose GPU computing promises to leverage the substantial parallelism of many-core GPUs to provide increased performance for applications that can be written to take advantage of GPU architectures. Numerous papers have demonstrated the impressive computing power of GPUs, and speedups of 1000x across a number of different types of applications have been reported over similar algorithms written for CPUs (for example Che et al. [1] and Ryoo et al. [2]). However, there is an ongoing discussion about the quantitative speedup that GPUs can provide over CPUs and whether real applications truly can benefit by two or more orders of magnitude improvement by utilizing GPU computing. Lee et al. [3] provide a compelling argument that fair comparisons are necessary; CPU and GPU performance evaluations are only valid when both have been fully optimized for their respective platforms, taking into consideration specific architectural features for each device and optimizing the kernels to take advantage of these as much as possible. We believe that even this argument lacks an important factor: the location of the data that is being processed before, during, and after kernel execution. As we show in this paper, it is not simply the speed at which a certain algorithm processes data that is important, but how that data has been used prior to, and

how it will be used after the kernel is run. GPU kernels are fast, and in many cases kernels are completed in the tens of milliseconds timeframe. Given this, the movement of data to and from the GPU becomes relevant, both because of the amount of data that needs to be moved, and because of the inherent bottleneck from the interconnect between the CPU and GPU. Without acknowledging (or rationalizing) the time for these data transfers, comparisons do not provide a true representation of the real speedup provided by a device. Our objective in this paper is not to challenge the benefits of using GPUs to accelerate computational work; rather, we would like to emphasize the importance of including data transfer overhead when making comparisons to CPU applications that have the data accessible on-chip.

It is one thing to acknowledge that for most GPU kernels, data must be moved onto the device prior to being used by the kernel, but it is more important to recognize that calculations performed on data are only worthwhile when the results of those calculations are further utilized. For example, if an array of integers has been sorted, the newly sorted array is useful only if the data will be analyzed by a follow-on function or algorithm. Unless the data will be used by another kernel on the GPU, some data must be returned to the CPU via the interconnect, whether it is the full data set, or a result based on the data (e.g., the result of summing an array of integers). Disregarding this aspect when reporting GPU kernel performance is deceptive, until such a time as there are shared memory CPU/GPU systems and there is no need to transfer data across a device-to-device interconnect for further processing or output.

We have two goals in this paper. The first is to demonstrate the necessity of reporting memory transfer overhead costs for CPU/GPU performance comparisons, and to show that many performance results have been misleading because they have disregarded this information. We have run benchmarks on eleven diverse GPU kernels and have included timing data for CPU/GPU memory transfers. Our results show that data transfer time can be as significant as kernel runtime, and that the faster the GPU, the more important it is to consider the data transfer overhead.

In most of the benchmarks, data is transferred onto and off of the GPU to be available for processing, and to be used after the kernel for further processing or output. In some cases, a large amount of data is transferred onto the GPU but only a

small amount of data is returned to the CPU. For example, the search kernel transfers a large text string to the GPU but only returns an integer value of the location of a found string. When the one-way transfer time is taken into account, the search kernel takes 2x longer than the GPU processing time, on average. In other cases, data does not need to be transferred to the GPU but a large amount of data is transferred back from the GPU. The Mersenne Twister kernel generates pseudorandom numbers on the device with a single integer as the only input data, but if the results are brought back to the CPU, this adds an average time of 7x over the GPU processing time alone. If instead the results remain on the GPU for use by another kernel, then there would be no memory transfer overhead at all. Finally, in the Monte Carlo kernel, only a small amount of data is transferred onto the device, and a small amount off; this kernel does many thousands of iterations, and the memory transfer is inconsequential. We chose what we believe to be “normal” usage cases, and we explain when data transfer assumptions can be modified, for instance when a kernel might likely be sandwiched between two other kernels such that the data is already present on the GPU.

Our second goal is to introduce a taxonomy that can be used for future comparisons that provides this information while at the same time being flexible enough for the myriad of situations for which a kernel might be used. We present five broad categories for the classification of kernels, based on the memory-transfer overhead such a kernel would experience in a typical usage case: Non-Dependent (ND), Dependent-Streaming (DS), Single-Dependent-Host-to-Device (SDH2D), Single-Dependent-Device-to-Host (SDD2H), and Dual-Dependent (DD). Depending on the actual usage scenario, most kernels could fit into more than one category, but we show that when reporting performance results it is important to include results from the most likely categories, and to explain situations when the different categories would be applicable.

II. RELATED WORK

It is well known that PCI Express bandwidth can cause a throughput bottleneck when a significant amount of data is transferred between a CPU and a GPU in a heterogeneous system. A number of researchers have discussed bandwidth troubles that can arise with frequent or poorly managed data movement between devices. Schaa and Kaeli [4] examine multiple GPU systems and acknowledge that unless a full working set of data can fit into the memory on a GPU, the PCI Express will be a bottleneck. Owens et al. [5] express similar concerns. Fan et al. [6], Cohen and Molemaker [7] and Dotzler et al. [8] all recommend rewriting algorithms to limit PCI Express transfers as much as possible. The aforementioned studies have served as our motivation for this paper, and we decided to quantify the memory transfer overheads for a number of benchmarks, and from the results we developed an overhead taxonomy for discussing the overhead in more detail.

Some researchers have made suggestions about how to mitigate data transfer overhead limitations. Al-Kiswany et

al. [9] describe StoreGPU, a distributed storage system that uses pinned, non-pageable memory on the host system to reduce the impact of data transfer. Gelado et al. [10] introduce an “Asymmetric Distributed Shared Memory” (ADSM) model that provides two types of memory updates (“Lazy” and “Rolling”) that determine when to move data on and off the GPU. Becchi et al. [11] implement a heterogeneous scheduler that takes memory transfer overhead into consideration, and chooses not to migrate data when the overhead is too great. Our work aims to provide further refinement and accuracy to the memory transfer overhead problem: by categorizing when memory transfer overhead is significant, strategies such as those just mentioned can improve.

An important GPGPU research area involves migrating database operations to the GPU, and moving all or portions of a database to the GPU involves a significant cost. Bakkum and Skadron [12] describe a SQL database GPGPU solution that demonstrates a 1.4x performance degradation when considering data transfer overhead in their results, and the entire database must also be located on the device. Volk et al. [13] implement a GPU database that also requires the entire database to be moved to the GPU in order to keep data transfer at a minimum. Additionally, they recognize that in order to benefit from a GPU solution, they must amortize the cost of moving the database to the device by performing multiple database operations. Govindaraju et al. [14] use a bitonic sort algorithm to perform database sort operations, and they specifically optimize their code to limit memory transfers between devices due to memory transfer overhead.

III. BENCHMARKS

Test Setup We chose a broad range of kernels that demonstrate a variety of memory-transfer overhead scenarios, and we chose GPUs and test CPU / PCI Express systems to show the differences that communications overhead can have depending on the speed of the GPU and the PCI Express setup in the system. All kernels were written in CUDA and were run on four platforms, as shown in Table I. Of note in Table I is that the GTX 480 GPU provides the fastest raw computing power but has the slowest bandwidth between the CPU and GPU, and the 330M (a laptop GPU) is the slowest card but provides one of the fastest bandwidth of any of the devices. For some of the benchmarks, the larger data sets would not fit onto the 330M, nor onto the 9800 GT. This is reflected in the figures as fewer bars; if there are only three bars in a set, the 330M did not run the benchmark for that data size, and if there are only two bars neither the 330M nor the 9800GT ran the benchmark for that data size.

The PCI Express Standard The PCI Express standard provides the motherboard interconnect between the CPU and the GPU, and all memory transfers between the CPU main memory and the GPU main (shared) memory flow over the PCI Express connection. Table I shows the PCI Express bandwidth for each CPU/GPU combination, and the bandwidth results were gathered using the CUDA SDK `bandwidthTest` application for each device. As we will show in the benchmark

GPU Type	Compute Capability	Cores	Memory (MB)	Clock (MHz)	Host-Dev BW (MB/s)	Dev-Host BW (MB/s)
Tesla C2050	2.0	480	3072	1150	2413.9	2359.2
GTX 480 (Fermi)	2.0	480	1024	1401	1428.0	1354.2
9800 GT	1.1	112	1024	1500	2148.8	1502.5
330 M	1.2	48	256	1265	2396.2	2064.7

TABLE I

GPUS TESTED. ALL GPUS ARE FROM NVIDIA AND RUN THE CUDA PROGRAMMING LANGUAGE. THE 330M GPU IS A LAPTOP GPU AND THE OTHERS ARE DESKTOP GPUS. “HOST-DEV” SHOWS TRANSFER TIMES FROM MAIN MEMORY TO GPU MEMORY, AND “DEV-HOST” SHOWS TRANSFER TIMES FROM GPU MEMORY TO MAIN MEMORY.

results, in many cases data transfer over this connection contributes the majority of the time to certain benchmarks executions. It is possible to asynchronously stream data over the PCI Express while data is being acted upon on the GPU, and the CUJ2K kernel we investigated demonstrates this feature. Memory transfers that take advantage of streaming must be used on page-locked CPU memory (because it is accomplished via direct memory access), and this adds to the complexity of adding the streaming function to an application. Furthermore, regardless of the amount of streaming used, most applications require that the full amount of data passes across the PCI Express interconnect during the kernel execution, and therefore the limiting factor for most applications is still this transfer.

There is one other type of memory transfer, called “zero-copy” that can be used to transfer data back and forth from the CPU to the GPU. It is a direct memory access function that also utilizes page-locked CPU memory, and the benefits of using zero-copy are that the GPU takes full control of the memory transfers (without CPU interaction), and can also utilize the CPU main memory as if it were the main memory on the GPU. Again, however, all of the data does have to eventually pass across the PCI Express in order for the GPU to manipulate it. As with streaming memory, the speedup is still limited by the PCI Express bandwidth.

It is important to note that faster GPUs are affected more by the PCI Express overhead, simply because the amount of time they spend completing kernels is proportionally smaller than for a slower GPU with a similar PCI Express bandwidth. As new GPUs provide greater compute performance, it becomes even more important to include this overhead when reporting performance comparisons.

The Kernels and Benchmark Results Table II shows the kernels we benchmarked, and the relative amount of data that each kernel transfers to the GPU and back from the GPU after the kernel completes. For example, the Sort algorithm we analyzed requires all of the data to be present on the GPU in order to run, but it does not send any data back to the CPU. The Mersenne Twister algorithm requires only an integer as input and creates pseudorandom numbers on the GPU, but a common case is to return them to the CPU after the kernel completes.

When running our benchmarks, we focused on testing a range of data sizes, culminating in finding the largest data set

Kernel	Data to GPU	Data to CPU
Sort	Large	None
Convolution	Large	Large
SAXPY	Large	Large
SGEMM	Large	Large
FFT	Large	Medium
Search	Large	Small
SpMV	Large	Medium
Histogram	Large	Small
Mersenne Twister	None	Large
Monte Carlo	Small	Small
CUJ2K	Large	Large

TABLE II

BENCHMARK KERNELS. THE DATA SIZE SENT TO THE GPU ASSUMES THAT THE DATA IS NOT ALREADY PRESENT ON THE GPU WHEN THE KERNEL LAUNCHES. THE DATA SIZE RETURNED TO THE CPU ASSUMES THE DATA WILL BE USED NEXT BY THE CPU.

that would allow the application to complete on a respective GPU. Having a large data size is important, particularly for applications that are not considered to be bandwidth bound. For instance, SGEMM, a compute bound application, still experiences a 50% slowdown for fast GPUs (in this case, the Tesla C2050 and the GTX 480) with large matrix sizes because the kernels complete so quickly.

1. **Sort** is a fast radix sorting method that has been shown to sort over one billion 32-bit keys on a GTX-480 GPU [15]. The benchmark we tested first places all of the keys into GPU main memory, runs the kernel, and then leaves the sorted array on the GPU for further processing. Figure 1 shows our benchmark results with these assumptions, and it is clear that the fast GPUs with a large data set are affected the most by the memory-transfer overhead. For the GTX 480 to sort 32MB of keys, 175ms was spent transferring the data to the GPU, while only 67ms was spent performing the sort; in other words, the overall time for the sort was 3.6 times longer than for the GPU-only portion. Discussions with the author of the application we benchmarked acknowledged this overhead, but also stated that it is just as likely that the sort kernel acts on data that was on the GPU for a previous kernel. While we agree with that assessment, the fact that there is a factor of over three between the two usage cases is significant. If the data had been returned to the CPU (a common usage case), the total application would have been over six times slower than the kernel alone.

2. **Convolution** is a widely used image filtering application

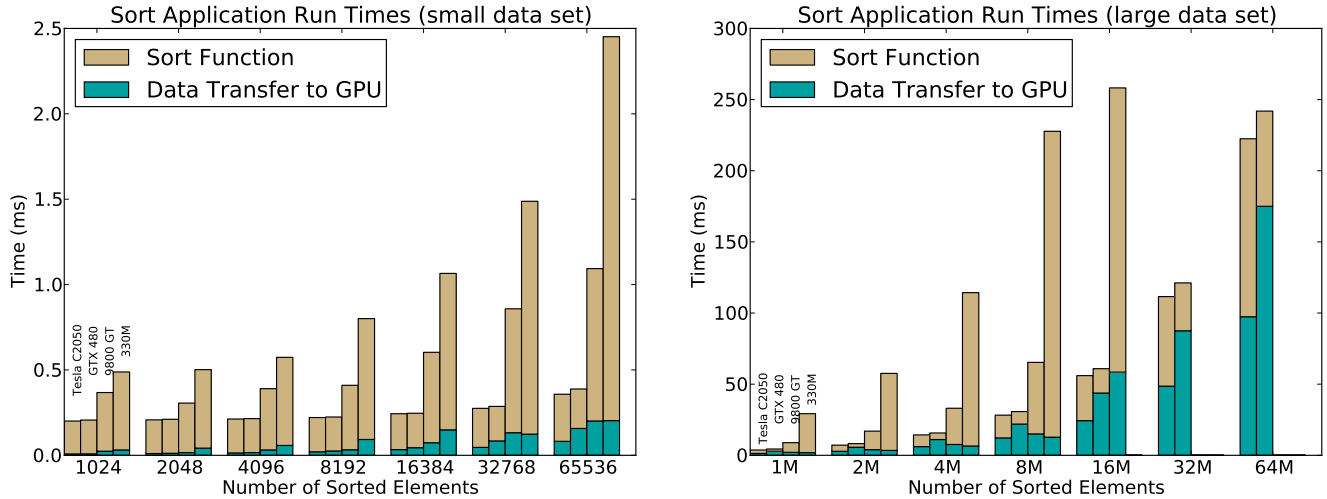


Fig. 1. Sort benchmark. Faster GPUs are affected more by the memory transfer overhead. For instance, when sorting 64M values, the application time on the GTX 480 is 3.6x slower than the kernel itself.

that can be used for smoothing, edge detection, and blur, among other functions. We benchmarked the separable convolution application example from the NVIDIA CUDA SDK toolkit [16]. A typical use for an image convolution on a GPU would transfer image data to the GPU, run the kernel, and transfer the convoluted image back to the CPU, and Figure 2 shows the results of the benchmark. For the large data set, the application takes more time to run on the fast GTX 480 than on the slower 9800 GT because the memory-transfer bandwidth is better for the 9800 GT.

3. **SAXPY** stands for “Scalar Alpha X Plus Y” and is a function in the Basic Linear Algebra Subprograms package. It is a straightforward multiply-and-add algorithm that is pleasingly parallel. NVIDIA provides an optimized, CUDA version of SAXPY in the CUBLAS package [17] and this is what we benchmarked. Figure 3 shows the results of the benchmark, and it is obvious that the memory-transfer overhead is overwhelming to the runtime for the application. On average, the kernel plus memory-transfer times took 43x longer than the kernel processing time alone.

4. **SGEMM** is the Single Precision General Matrix Multiply algorithm. The CUBLAS package also includes SGEMM, and we ran our benchmark on the CUBLAS application. Like SAXPY, SGEMM is pleasingly parallel, but because it is an n^3 algorithm, it performs more calculations on the matrices than SAXPY, and Figure 4 shows that the memory-transfer overhead is not as overwhelming to the application as for SAXPY, especially as the data set increases. However, the run time for the Tesla C2050 is still almost twice as slow for the largest data set as it would be without the memory-transfer overhead.

5. **FFT** is the Fast Fourier Transform algorithm, which transforms signals in the time domain into the frequency domain. NVIDIA provides the CUFFT library [18], which we benchmarked. Figure 5 shows that for large data sets, there

is more than 100% overhead for data transfer on fast devices. FFT returns less data than is passed into the function, so it does take slightly less time for the transfer back from the GPU. It is still apparent that the fast GPUs are constrained significantly by the memory-transfer overhead.

6. **Search** is a simple textual search for a short random lowercase alphabetic string in a long random string of lowercase alphabetic characters. The code we benchmarked was based on an example from an online supercomputing performance and optimization analysis tutorial [19]. For each benchmark, we measured the total time to find 1000 different search strings in a certain length sample text. As the sample text gets larger, the likelihood for finding the search string increases, and as Figure 6 shows, when the size of the sample text reaches roughly two million characters, the average search time levels out. The Tesla C2050 incurs a memory-transfer penalty of 2.5x at this point, and the GTX 480 incurs a penalty of 5x. The GT 9800, with CUDA compute capability 1.1*, has comparably poor performance compared to all three other GPUs because the benchmark uses atomic operations that were significantly optimized for later compute capabilities.

7. **SpMV**, or Sparse Matrix Vector multiplication, is an important sparse linear algebra application, and it is a bandwidth-intensive operation when matrices do not fit into on-chip memory. We benchmarked an implementation described in Bell and Garland [20] for the coordinate (or “triplet”) format that has a storage size proportional to the number of non-zeros in the sparse matrix. Figure 7 shows the results for three different matrices (suggested by Bell and Garland [20] and used originally for the work in Williams et al. [21]). The application transfers much more data to the GPU than it returns, and thus the data transfer time is weighted to the amount transferred to the GPU. In the case of the largest

*The *compute capability* of a device denotes the CUDA features an NVIDIA GPU supports; it is akin to a version number.

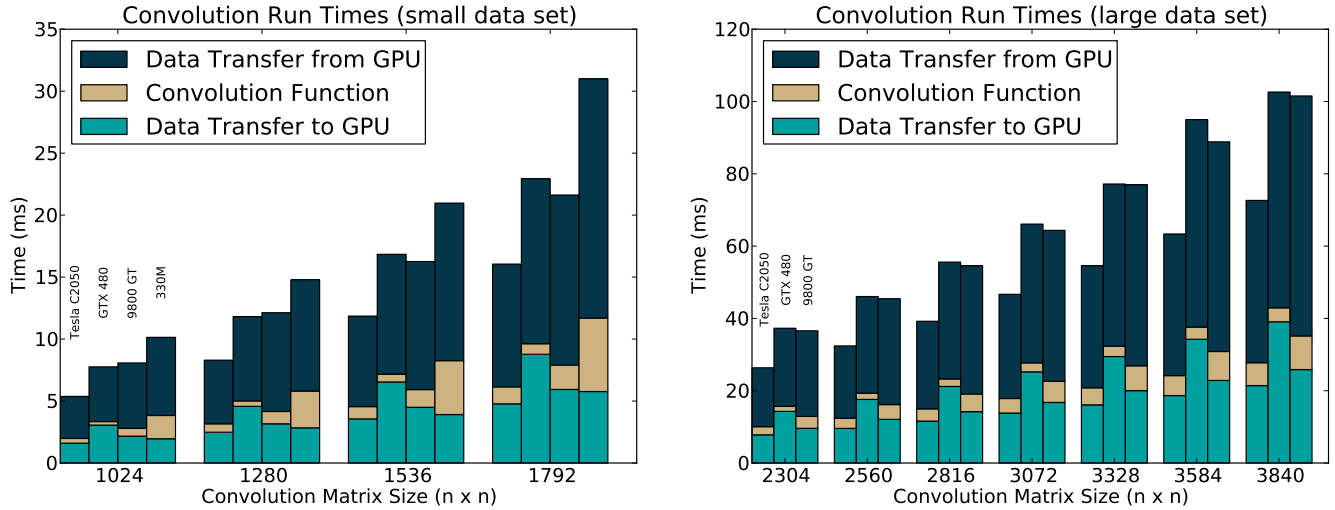


Fig. 2. Convolution benchmark. The benchmark time is dependent on data transferred to the GPU and also on data transferred back to the CPU. Note that the slower 9800GT GPU has a faster overall run time than the much faster GTX 480 because of the slower transfer times on the GTX 480.

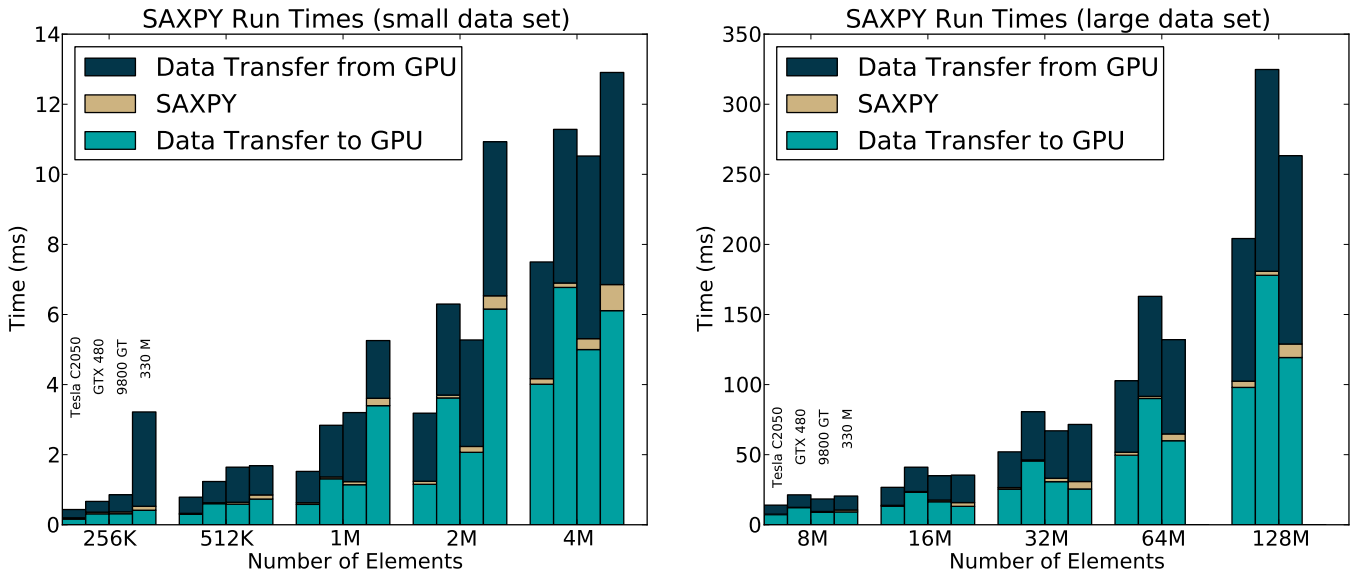


Fig. 3. SAXPY benchmark. Because the CUBLAS version of SAXPY is well optimized and pleasingly parallel, the memory-transfer overhead comprises almost all of the application run time. Again, for large data sets the faster GTX 480 performs worse for the overall benchmark than the slower 9800 GT.

two data sets, `mc2depi.mtx` and `webbase-1M.mtx`, the device memory-transfer bandwidth completely dictates the time for the application, and the less powerful 330M GPU completes the application faster than the GTX 480 simply because it has a higher CPU-GPU bandwidth.

8. **Histogram** is an image processing algorithm that combines a stream of pixel light values into a series of bins that represent the distribution of light across an image. We benchmarked the CUDA SDK histogram application that computes both a 64-bin histogram and a 256-bin histogram on a set of data [22]. The histogram application sends a large amount of data to the GPU, but simply returns either a 64-byte or 256-

byte array representing the bins. Therefore, as can be seen in Figure 8, the memory-transfer overhead for the data sent to the GPU is significant, while the memory-transfer overhead for the data sent back is not.

9. **Mersenne Twister** is an algorithm for generating pseudorandom numbers. There is no transfer of data to the GPU (except a single integer seed) and the algorithm produces the values solely on the device. Figure 9 shows that the only memory-transfer overhead comes from returning the results to the CPU.

10. **Monte Carlo** is an algorithm that repeatedly and randomly samples a function many times, averaging the results.

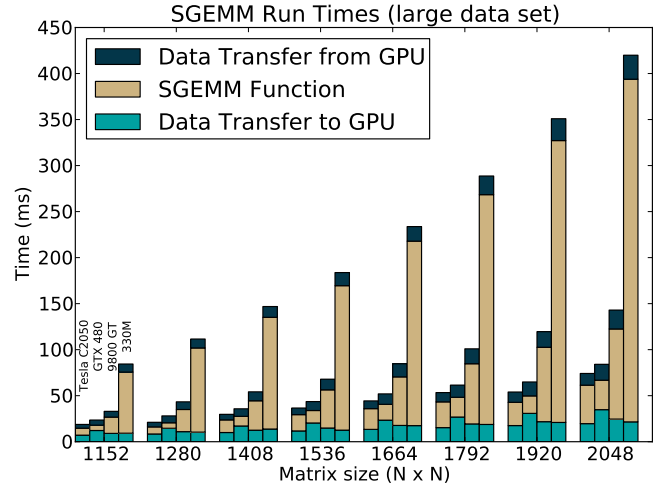
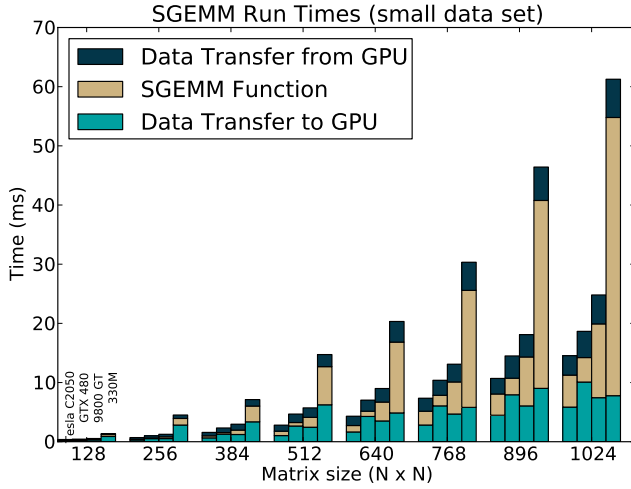


Fig. 4. SGEMM benchmark. SGEMM is another pleasingly parallel application, but the n^3 algorithm in the kernel does spend a significant amount of time relative to the memory-transfer overhead.

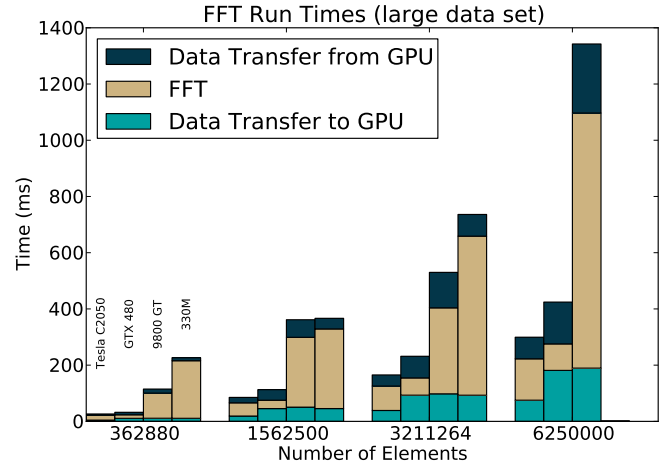
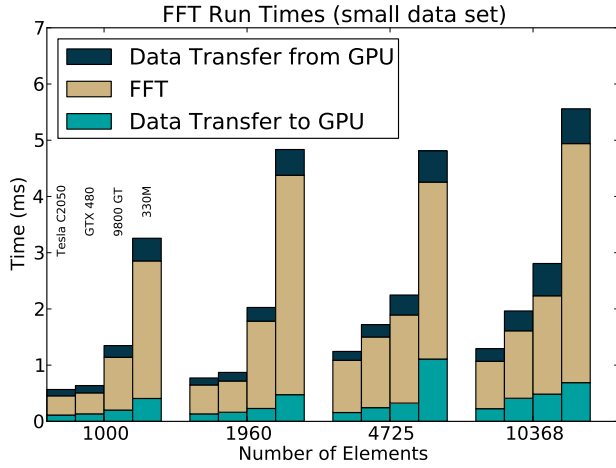


Fig. 5. FFT benchmark. FFT produces a similar profile to SGEMM, and the fast GPUs incur a time penalty for memory-transfer of over 100% for the largest data set.

We benchmarked a financial option pricing application from the CUDA SDK [23], with 2048 options, with 256K to 32M sample paths. In terms of this study, Monte Carlo is almost perfectly suited to run on a GPU, as only 32KB of data is passed to the device and 16KB is returned to the GPU, regardless of how many sample paths are computed. Figure 10 shows that in many cases the memory-transfer overhead is inconsequential compared to the run time of the algorithm itself.

11. **CUJ2K** is a CUDA application that converts BMP images to the JPEG 2000 format [24]. By default it runs in asynchronous streaming mode, moving data on and off the GPU at the same time it encodes data already on the device. We measured kernel times for both streaming and non-streaming conversions, as shown in Figure 11. For CUJ2K, asynchronous streaming improves performance when more

than one image is processed by the application, and for the Tesla C2050, the speedup when converting 16 images is $5x$ over the non-streaming version. Utilizing asynchronous memory transfers is the optimal solution if data must be moved between devices, as computation and data transfer happen concurrently.

IV. A TAXONOMY FOR MEMORY TRANSFER OVERHEAD

The previous section demonstrated that data transfer times are often a critical aspect of overall GPU application times. We have identified five different types of application usages that categorize the dependence on memory transfer for GPU kernels. Broadly, the categories indicate whether there is a significant dependence on transferring data to or from the GPU, which is important for both CPU/GPU comparison reporting and also for informing decisions about scheduling and other heterogeneous computing scenarios.

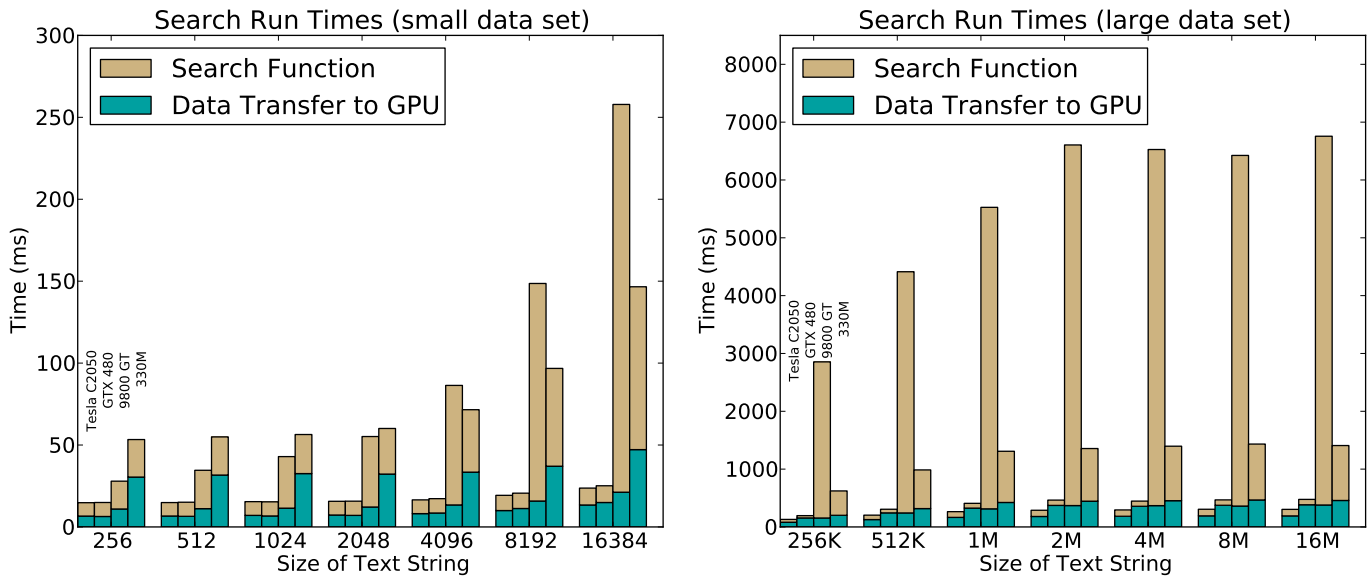


Fig. 6. Search benchmark. The bars represent the time to search for 1000 random 4-character strings in random text string that increases in size. The text string is transferred to the GPU for each iteration. The 9800 GT's poor performance is due to the lack of optimized atomic operations in GPUs with compute capabilities less than 1.2.

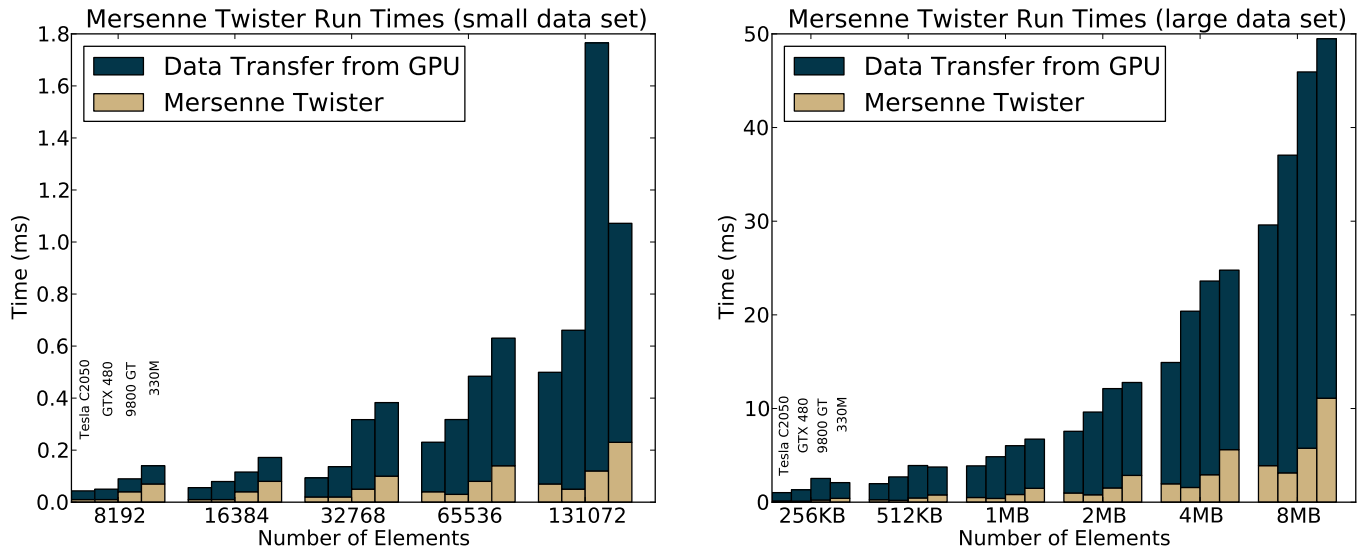


Fig. 9. Mersenne Twister benchmark. All pseudorandom numbers are generated on the GPU, so the only memory-transfer overhead comes from transferring the results back to the CPU.

Categories of Application Kernels

GPU kernels fit into five specific categories:

- 1) **Non-Dependent (ND)** Those that do not depend on data transfer to or from the GPU, or the dependence is extremely small (a single value as a seed, for instance, or an integer returned as a result).
- 2) **Dependent-Streaming (DS)** Those that are dependent on data transfer to or from the GPU but hide this dependency with asynchronous streaming memory.

- 3) **Single-Dependent-Host-to-Device (SDH2D)** Those that depend on data transfer to the GPU.
- 4) **Single-Dependent-Device-to-Host (SDD2H)** Those that depend on data transfer from the GPU.
- 5) **Dual-Dependent (DD)** Those that depend on data transfer to *and* from the GPU.

It is possible for a kernel to fit into more than one category for different usage cases; for instance a kernel that can either run independently on a set of data, or can run as a part of a sequential set of kernels on the same set of data would be in

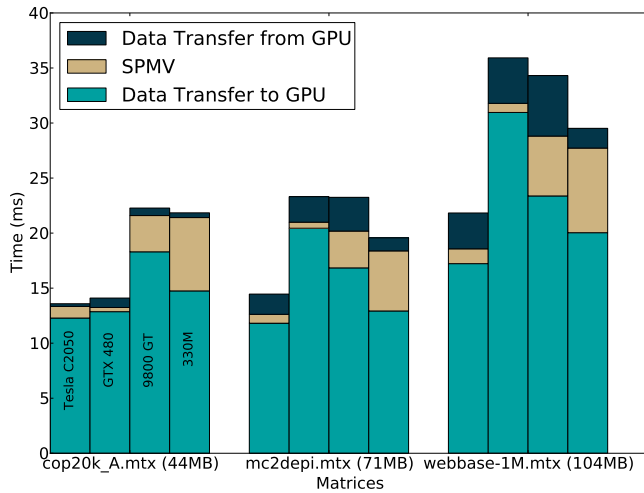


Fig. 7. The SpMV benchmark was run on three sparse matrices stored in memory in coordinate format. The matrix size is in parentheses next to each label.

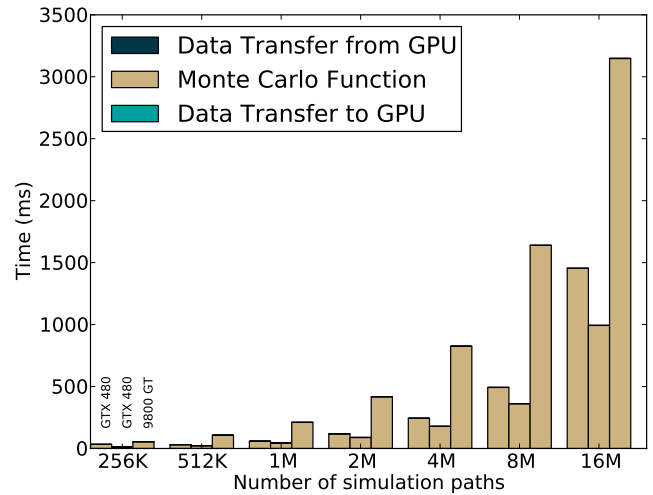


Fig. 10. Monte Carlo benchmark. Regardless of the number of sample paths, only 32KB of data is transferred to the GPU and only 16KB is returned to the CPU. The values are insignificant compared to the overall run time of the application.

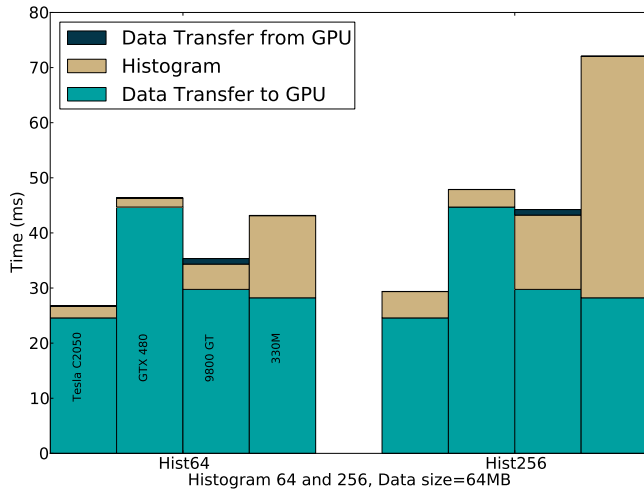


Fig. 8. Histogram benchmark. A large amount of data is sent to the GPU, but only one small vector (64-bytes or 256-bytes) is returned to the CPU.

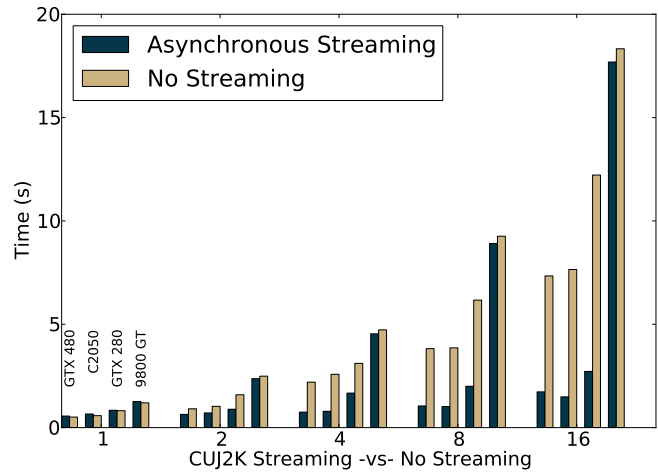


Fig. 11. CUJ2K. The asynchronous streaming version of the application performs up to $5\times$ better for faster devices. The x-axis denotes the number of BMPs that were converted to JPEG-2000 at a time. Each BMP was 2048×1361 pixels and 24-bit color. *Note: the 330M was replaced with a GTX 280 for this experiment.*

two different categories depending on the scenario. It is up to whomever is presenting the performance data to indicate which categories make the most sense for the kernel in question (as we have done in Table III).

Kernels that fit into the ND category process data that is already on the GPU and either do not return the data to the CPU or return a single value or a small number of values (e.g., the integer location of search). If a sequence of kernels all act on a single set of data that is loaded onto the GPU initially and moved back from the GPU after all the kernels have run, all kernels except for the first and the last will fall into this category. The ND category provides the GPU-speedup proponents with their best results, and is indeed where most of the performance comparisons are generated.

Some applications that would not generally fit into the ND category can show similar results if they are able to take advantage of streaming memory to asynchronously transfer data to the GPU while it is simultaneously acting on part of the data. NVIDIA's CUDA has had the ability to stream data since compute capability 1.1 [25], and the OpenCL API supports asynchronous data transfer as well [26]. An application must specifically be written to synchronize the pipelining of such data, and there are some overheads to this synchronization.

Applications that fit into the DS category include those that implement streams to overlap data transfers with computation. Wu et al. [27] implement a GPU version of the K-Means clustering algorithm and demonstrates the use of streaming

to hide data transfer overhead. They focus on huge data sets with billions of points that do not fit onto the GPU at once.

Kernels that fit into either the SDH2D or the SDD2H categories have similar overhead in that they must transfer data either to or from the GPU, but not both. Examples of the former would be a search algorithm that moves data onto the GPU, performs the search, and returns an integer value for the location of the search term, if found. An example of the latter would be Mersenne Twister, which produces a set of pseudorandom numbers on the GPU and returns them to the CPU. The difference between the two categories is simply at which point the memory transfer overhead is applied, at the beginning or at the end. For example, if a heterogeneous scheduler knows that there will be a memory transfer dependency at the beginning of a kernel launch, it knows that the CPU memory will be busy immediately, and it can take that into consideration for scheduling decisions.

Kernels that fit into the DD category have the most overhead to consider. Any kernel that has the sole purpose to act on data that first needs to be moved to the GPU and then moved back to the CPU main memory will be in the DD category, and this includes most of the types of kernels reported on in the literature. However, DD kernels also have the most potential to be optimized to either fit into a different category, either by streaming data, or by being incorporated into a sequence of kernels that all act on the same data.

As a rule, kernels can move between categories depending on their eventual use, as demonstrated with kernels such as sort that may work with data *in situ* as part of a larger application (or as part of a pipeline of kernels acting on a set of data), or as a standalone sorting function where the only use for the GPU is to perform the sort. GPU speedup comparisons should discuss the types of cases that would be relevant for the kernel being measured, and if there is a proposed general-use case that should be mentioned as well.

V. THE BENCHMARKS CATEGORIZED

Table III shows the benchmarks from section III, and the categories they fit in, based upon the benchmark results. The table also lists the secondary categories that each benchmark is likely to fall into if it was used in a more general application. For example, the search kernel we analyzed loads the entire array to be searched into GPU memory and then performs the search on the entire data set. It could be improved with a streaming memory implementation that searches the data as it arrives onto the GPU. Virtually all kernels would fit into the ND category if they were part of a sequential set of kernels and were run in between two other kernels. Similarly, if those same kernels were either the first or last of a sequential set of kernels they would either fall into the SDH2D or SDD2H categories.

Half of the applications we investigated fit into the DD category (Convolution, SAXPY, SGEMM, FFT, and SpMV). There are significant memory transfers both to the GPU before the kernels run and back to the CPU after the kernels finish. We can also envision most these applications in the ND secondary

Application	Category	Secondary Category
Sort	SDH2D	ND
Convolution	DD	ND
SAXPY	DD	ND
SGEMM	DD	ND
FFT	DD	SDH2D
Search	SDH2D	DS
SpMV	DD	DS
Histogram	SDH2D	DS
Mersenne Twister	SDD2H	DS
Monte Carlo	ND	–
CUJ2K	DS	DD

TABLE III
DEFAULT AND SECONDARY MEMORY TRANSFER CATEGORIES FOR EACH BENCHMARK. MOST APPLICATIONS FIT INTO THE DUAL-DEPENDENT CATEGORY. THE SECONDARY CATEGORY LISTS A MOST-LIKELY SCENARIO FOR A MORE GENERALIZED INCLUSION OF THE BENCHMARK IN A REAL APPLICATION.

category. We placed FFT in the SDH2D secondary category because there are many signals processing applications that gather data off of a sensor in the time domain, perform FFT to move the data into the frequency domain, and then use the data directly, which could also be done on the GPU.

It is clear from the benchmark results that the Sort, Search, and Histogram applications fit into the SDH2D as they all need data moved to the GPU in order to begin their kernels, but they do not return much data. StoreGPU [9] fits into the SDH2D category, and the data transferred onto the GPU is many magnitudes greater than the amount that is transferred off. We have listed ND as the secondary category for Sort because there are many cases where a set of values needs to be sorted as part of a larger sequence of operations. Both Search and Histogram would benefit from streaming, and we have included DS as their secondary categories.

Mersenne Twister is the only application we benchmarked that fits into the SDD2H as a primary category. It would benefit from streaming the data back to the CPU, so we placed it into the DS secondary category. If the data was left on the GPU for further processing by other kernels, it would fit into the ND category. As seen from Figure 10, Monte Carlo has no significant dependencies for large data sets and fits nicely into the ND category. For some applications, such as the database applications listed in Section II, there is a large initial memory transfer that implies a SDD2H category, but the data then resides on the device for future kernel calculations, indicating that a long-term ND categorization would be appropriate. There are many situations in which a kernel forms a part of a larger set of calculations, and the data can reside on the GPU throughout each kernel. In this case, the kernels in the middle of this pipelined scenario would fall in the ND category. Hoberock et al. [28] utilize a sort kernel that sorts data already present on the device and does not return it after the kernel is complete.

CUJ2K utilizes asynchronous streaming memory to partially hide data transfer between the CPU and the GPU, and therefore it has a DS classification. It can also run as a non-streaming application, so its secondary category is DD.

VI. CPU/GPU REPORTING RECOMMENDATIONS

We now propose a methodology for reporting on GPU performance based on our analysis of CPU/GPU memory-transfer overhead. Performance gains that include this information will provide a truer picture of not only the real speed increases that can be expected in a heterogeneous system, but will also yield more information for scheduling decisions and for more finely targeted optimizations.

Describing the “Typical Usage” of an Application: The value of a good algorithm comes when it is actually used. `Histogram` is widely used in image processing, and one common use is to take a histogram of an image and then find the minimum or maximum threshold values to use for edge detection. Therefore, a typical use would move image data onto the GPU, where a histogram kernel precedes a threshold algorithm, which could also occur on the GPU. Therefore, a typical use for `Histogram` would include moving data onto the GPU and leaving the data there for a follow-on algorithm. This is important because it demonstrates that there is a dependency for memory-transfer onto the device but not off of it. If a typical usage reduces the dependency to move data from the CPU to the GPU or vice-versa, this is important to report. All typical usage cases should be included; e.g., if it is likely that a sorting algorithm will act on data that is already present on the GPU and the data will remain on the GPU, but the algorithm might also be the last kernel to run before data is transferred back to the CPU, this should be noted.

Using the Taxonomy: Our proposed taxonomy provides a broad amount of information about otherwise hidden memory-transfer dependencies of GPU kernels. `Mersenne Twister` is an excellent example: labeling it as `SDD2H` indicates that there is no dependence on data transferred to the GPU, but that the data will come back to the CPU for further use. Using the taxonomy is even more critical when there are different usage cases, so that someone wishing to use an algorithm can understand the performance based on how it will be used. If `Mersenne Twister` is going to return pseudorandom numbers to the CPU that will immediately be used, it would be nice to know that the algorithm is, in general, `SDD2H` but can also be coded as `DS` to indicate that it can be streamed back to the CPU as well.

Indicating the Amount of Data that will be Moved: We purposefully developed the memory-transfer overhead taxonomy to be broad in order to limit the number of categories. However, there might be cases where the amount of data transferred does not allow an algorithm to fit nicely into a specific category. A `Histogram` that includes many more than 256 bins would be an example, and it might be the case that a histogram on a small amount of data with many bins might be better described as `DD` instead of `SDH2D`. Our recommendation is to simply indicate that this is a borderline case, or to ensure that the typical use description is robust.

VII. HETEROGENEOUS SCHEDULING RAMIFICATIONS

Dynamic desktop heterogeneous scheduling is a developing research area in which a scheduler makes a decision about

how to partition work between the heterogeneous processors in a system. One key piece of information necessary when trying to make a decision about whether it is worthwhile to launch a GPU kernel or to perform the same function on the CPU is the projected overall time for the kernel, including all data transfers. As we have shown in this paper, profiling the kernel on the GPU without including the data transfer times to and from the GPU overestimates the GPU speedup, and would lead to poor scheduling decisions if used. The taxonomy we propose provides a way for a scheduler to obtain the extra information about memory transfer overhead but also allows flexibility for the way a kernel will be utilized in an overall application.

We envision that a CPU/GPU scheduler will need to apply a taxonomic category to any kernel that it might schedule on a GPU. If the kernel is categorized as `SDH2D` and the data is only on the CPU and not on the GPU, the scheduler can calculate the added overhead time for the kernel based on the amount of data that needs to be transferred. If, on the other hand, the data is already on the GPU, the scheduler does not need to apply this overhead, and will make its decision of whether or not to launch the kernel based on this extra information.

VIII. CONCLUSION

In this paper, we have shown that traditional reporting of CPU/GPU performance comparisons falls short because it does not adequately describe the memory-transfer overhead that applications run on a GPU incur in typical use cases. We benchmarked eleven diverse applications for a number of different data sizes on a range of heterogeneous platforms, and showed that the memory-transfer overhead adds a significant amount of time to virtually all applications. For some applications with large data sets, the memory-transfer overhead combined with the kernel time took longer than 50x the GPU processing time itself. However, the amount of overhead can vary drastically depending on how a GPU kernel will be used in an application, or by a scheduler. We therefore proposed a taxonomy for labeling applications to indicate these dependencies, and we made recommendations on how to utilize this taxonomy for future performance reporting. Future work will include using this information to inform scheduling decisions about whether to run kernels on a GPU or on the CPU.

ACKNOWLEDGMENTS

This work was supported in part by NSF grants CNS-0747273, CSR-0916908, and CNS-0964627, SRC GRC task number 1790.001, and a grant from Microsoft. We would also like to thank Tor Aamodt, Balaji Dhanasekaran, and the anonymous reviewers for their helpful comments and suggestions.

REFERENCES

- [1] S. Che, M. Boyer, J. Meng, D. Tarjan, J. Sheaffer, and K. Skadron, “A performance study of general-purpose applications on graphics processors using CUDA,” *Journal of Parallel and Distributed Computing*, vol. 68, no. 10, pp. 1370–1380, October 2008.

- [2] S. Ryoo, C. I. Rodrigues, S. S. Baghsorkhi, S. S. Stone, D. B. Kirk, and W.-m. W. Hwu, "Optimization principles and application performance evaluation of a multithreaded GPU using CUDA," in *13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2008, pp. 73–82.
- [3] V. Lee, C. Kim, J. Chhugani, M. Deisher, D. Kim, A. Nguyen, N. Satish, M. Smelyanskiy, S. Chennupaty, and P. Hammarlund, "Debunking the 100x GPU vs. CPU myth: An evaluation of throughput computing on CPU and GPU," in *37th Annual International Symposium on Computer Architecture*, Saint-Malo, France, June 2010.
- [4] D. Schaa and D. Kaeli, "Exploring the multiple-GPU design space," in *International Parallel and Distributed Processing Symposium.*, May 2009, pp. 1–12.
- [5] J. Owens, M. Houston, D. Luebke, S. Green, J. Stone, and J. Phillips, "GPU computing," *Proceedings of the IEEE*, vol. 96, no. 5, pp. 879–899, May 2008.
- [6] Z. Fan, F. Qiu, A. Kaufman, and S. Yoakum-Stover, "GPU cluster for high performance computing," in *ACM/IEEE Conference on Supercomputing*, Pittsburgh, PA, November 2004, pp. 47–58.
- [7] J. Cohen and M. Molemaker, "A fast double precision CFD code using CUDA," in *Parallel Computational Fluid Dynamics: Recent Advances and Future Directions*, Moffett Field, CA, May 2009, pp. 414–429.
- [8] G. Dotzler, R. Veldema, and M. Klemm, "JCUDAmp: OpenMP/Java on CUDA," in *3rd International Workshop on Multicore Software Engineering*, May 2010, pp. 10–17.
- [9] S. Al-Kiswany, A. Gharaibeh, E. Santos-Neto, G. Yuan, and M. Ripeanu, "StoreGPU: exploiting graphics processing units to accelerate distributed storage systems," in *Proceedings of the 17th International Symposium on High Performance Distributed Computing*, Boston, MA, June 2008, pp. 165–174.
- [10] I. Gelado, J. E. Stone, J. Cabezas, S. Patel, N. Navarro, and W.-m. W. Hwu, "An asymmetric distributed shared memory model for heterogeneous parallel systems," in *Architectural Support for Programming Languages and Operating Systems*, Pittsburgh, PA, March 2010, pp. 347–358.
- [11] M. Becchi, S. Byna, S. Cadambi, and S. Chakradhar, "Data-aware scheduling of legacy kernels on heterogeneous platforms with distributed memory," in *SPAA: Proceedings of the 22nd ACM Symposium on Parallelism in Algorithms and Architectures*, June 2010, pp. 82–91.
- [12] P. Bakkum and K. Skadron, "Accelerating SQL database operations on a GPU with CUDA," in *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*. ACM, 2010, pp. 94–103.
- [13] P. Volk, D. Habich, and W. Lehner, "GPU-based speculative query processing for database operations," in *Proceedings of the 1st International Workshop on Accelerating Data Management Systems Using Modern Processor and Storage Architectures*, Singapore, September 2010.
- [14] N. Govindaraju, J. Gray, R. Kumar, and D. Manocha, "Gputerasort: High performance graphics co-processor sorting for large database management," in *Proceedings of the ACM SIGMOD International Conference on Management of Data*, June 2006, pp. 325–336.
- [15] D. Merrill and A. Grimshaw, "Revisiting sorting for GPGPU stream architectures," University of Virginia, Department of Computer Science, Charlottesville, VA, Tech. Rep. CS2010-03, 2010.
- [16] V. Podlozhnyuk, "Image convolution with CUDA," *NVIDIA Corporation white paper*, June, vol. 2097, no. 3, 2007.
- [17] NVIDIA, "CUBLAS library 3.1," http://developer.download.nvidia.com/compute/cuda/3_1/toolkit/docs/CUBLAS_Library_3.1.pdf, 2010. Accessed February 21, 2011.
- [18] —, "CUFFT library 3.0," http://developer.download.nvidia.com/compute/cuda/3_0/toolkit/docs/CUFFT_Library_3.0.pdf, 2010. Accessed February 21, 2011.
- [19] "Search algorithm with CUDA," <http://supercomputingblog.com/cuda/search-algorithm-with-cuda/>, Accessed: July 28, 2010.
- [20] N. Bell and M. Garland, "Implementing sparse matrix-vector multiplication on throughput-oriented processors," in *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, Portland, OR, November 2009, pp. 1–11.
- [21] S. Williams, L. Oliker, R. Vuduc, J. Shalf, K. Yelick, and J. Demmel, "Optimization of sparse matrix-vector multiplication on emerging multicore platforms," *Parallel Computing*, vol. 35, no. 3, pp. 178–194, 2009.
- [22] V. Podlozhnyuk, "Histogram calculation in CUDA," http://www.nvidia.com/object/cuda_sample_data-parallel.html, 2007.
- [23] V. Podlozhnyuk and M. Harris, "Monte carlo option pricing," *nVidia Corporation Tutorial*, 2008.
- [24] N. Fürst, A. Weiß, M. Heide, S. Papandreou, and A. Balevic, "CUJ2K," <http://cuj2k.sourceforge.net/cuj2k.html>, Accessed February 21, 2011.
- [25] NVIDIA, "CUDA programming guide, version 3.1," http://developer.download.nvidia.com/compute/cuda/3_1/toolkit/docs/NVIDIA_CUDA_C_ProgrammingGuide_3.1.pdf, July 2010. Accessed February 21, 2011.
- [26] J. E. Stone, D. Gohara, and G. Shi, "OpenCL: A parallel programming standard for heterogeneous computing systems," *Computing in Science and Engineering*, vol. 12, pp. 66–73, 2010.
- [27] R. Wu, B. Zhang, and M. Hsu, "Clustering billions of data points using GPUs," in *Proceedings of the Combined Workshops on UnConventional High Performance Computing Workshop Plus Memory Access Workshop*, 2009, pp. 1–6.
- [28] J. Hoberock, V. Lu, Y. Jia, and J. C. Hart, "Stream compaction for deferred shading," in *Proceedings of the Conference on High Performance Graphics 2009*, August 2009, pp. 173–180.