# An Improved Abstract GPU Model with Data Transfer

Thomas C. Carroll*, Prudence W.H. Wong
Department of Computer Science, University of Liverpool, UK
Email:{thomas.carroll, pwong}@liverpool.ac.uk

***Abstract*—GPUs are commonly used as coprocessors to accelerate a compute-intensive task, thanks to their massively parallel architecture. There is study into different abstract parallel models, which allow researchers to design and analyse parallel algorithms. However, most work on analysing GPU algorithms has been software based tools for profiling a GPU algorithm. Recently, some abstract GPU models have been proposed, yet they do not capture all elements of a GPU. In particular, they miss the data transfer between CPU and GPU, which in practice can cause a bottleneck and reduce performance dramatically. We propose a comprehensive model called Abstract Transferring GPU which to our knowledge is the first abstract GPU model to capture data transfer between CPU and GPU. We show via experiments, that existing abstract GPU models cannot sufficiently capture all of the actual running of a GPU algorithm time in all cases, as they do not capture data transfer. We show that by capturing data transfer with our model, we are able to obtain more accurate predictions of the GPU algorithm actual running time. It is expected that our model helps improve design and analysis of heterogeneous systems consisting of CPU and GPU, and will allow researchers to make better informed implementation decisions, as they will be aware how data transfer will affect their programs.**

## I. Introduction

Graphics Processing Units (GPU) are consumer grade pieces of hardware designed for high throughput computation associated with graphical operations. Early attempts at general purpose programming on GPU (GPGPU) resulted in manipulation of graphical libraries to perform the tasks, yet specialised frameworks such as OpenCL [1] and CUDA [2] have now taken their place. GPGPU is being increasingly used in scientific applications. Research in GPGPU has focused on performance estimation [3][4][5][6][7], as opposed to theoretical analysis. Recently, some abstract GPU models have been proposed [8] [9], yet they do not capture data transfer between CPU and GPU.

In this paper, we introduce a new model called Abstract Transferring GPU (ATGPU); an abstract model for design and analysis of GPU algorithms, which is an extension of previous models. Our model captures data transfer between CPU and GPU. To our knowledge, ATGPU is the first abstract GPU model to capture this. We demonstrate the usefulness of our model by analysing some computational problems on the model, and show how capturing the data transfer can improve the prediction of actual running time for a GPU algorithm, when compared to existing models.

*Thomas C. Carroll is supported by an EPSRC Studentship.

In this section, we discuss the modern CUDA GPU, followed by how classical parallel models relate to the GPU. We continue with discussing recent progress in modelling GPUs, and work related to CPU GPU data transfer. We then identify scope for improving abstract modelling of GPUs, and detail our contribution.

### *A. Modern CUDA GPUs*

The GPU is made up of many low powered cores, arranged in groups on streaming multiprocessors. A GPU has several streaming multiprocessors. Each streaming multiprocessor also has shared memory, divided into banks, accessible only to cores on the streaming multiprocessor. Part of the shared memory is reserved as register space for each core. The GPU has off-chip global memory split into equal sized memory blocks. Global memory is accessible by all cores on the GPU and by the CPU and is normally gigabytes in size.

A GPU program, also known as a kernel, contains a set of instructions for a single thread. The GPU launches many instances of the kernel for a large number of threads. Threads are grouped into thread blocks, with a thread block executing on a single streaming multiprocessor, working in parallel. Smaller groups of threads called a *warp* are run in SIMD fashion on the cores. There can be multiple warps within a thread block. Warps of the thread block can be synchronised using barrier operations and can only communicate via shared memory. Warps can only communicate outside the thread block via global memory. The list of instructions for each warp are placed into an instruction pool, and are scheduled to execute only when all dependencies are met and when cores become available on the streaming multiprocessor.

In a memory access instruction, the warp requests data from addresses and then waits until the request is resolved. Shared memory access is resolved almost as fast as register access, provided that the addresses are in distinct banks. If addresses are in non-distinct banks, then a *bank conflict* occurs, and requests serialize. Global memory access instructions fetch entire memory blocks at a time, and take much longer to resolve than shared memory access. If addresses reside in multiple memory blocks, then multiple memory blocks are fetched. Coalesced memory access is where a warp only requests addresses residing in the same memory block.

Whilst a warp waits for a memory request, other warps execute on the cores of the streaming multiprocessor, so that cores do not lie idle. If this happens throughout the duration of the memory request, the wait time is hidden by operations of other warps. This is known as *latency hiding*. A streaming multiprocessor can execute multiple thread blocks, provided there is enough shared memory space to hold all required data. The number of thread blocks resident on a streaming multiprocessor is known as *occupancy*. A higher occupancy increases the size of the instruction pool, meaning the amount of latency hiding could be increased.

## B. Classical Parallel Models

The *Parallel Random Access Machine* (PRAM) [10] is a shared memory model, containing asynchronous processors and shared memory. The PRAM does not include a memory hierarchy. Therefore, the PRAM misses important components needed for modelling or analysing GPU computation.

The *Bulk Synchronous Parallel Model* (BSP) [11] is a distributed memory model, consisting of a machine and a cost function. The machine contains processors with their own private memory and all processors are connected to each other. A processor accesses its own memory with low latency, and accesses another processor's memory with higher latency. Algorithms are executed in rounds of computation, communication, and synchronisation. Algorithms are analysed with a cost function, which is a function of the longest computation at each round, the number and size of communications at each round, and the cost of synchronisations. The lack of shared memory between processors and the allowed communication between processors means the BSP model does not capture everything to effectively model a GPU.

Tiskin [12] proposed the *Bulk-Synchronous Parallel Random Access Machine* (BSPRAM), consisting of processors with fast private memory, and shared memory accessible to all. The execution runs in rounds, similar to BSP. The BSPRAM is closer to GPU than PRAM and BSP, yet the architecture is not similar enough to the GPU as the notion of a warp is not present.

The *Parallel External Memory Model* (PEM) [13] contains processors and a formal memory hierarchy; each processor has private memory, and there is main memory accessible to all. Both memories are partitioned into equal sized memory blocks. Memory transactions transfer entire memory blocks, which is similar to global memory access in GPU. Algorithms are analysed on the number of memory transactions. The lack of shared memory for groups of processors and the lack of a warp means this model is not suitable for GPU modelling.

## C. Recent Progress on Modelling GPU

Recent progress on modelling GPU has come in two main areas: abstract modelling, and predictive tools. Two prominent abstract models for GPGPU are *Abstract GPU (AGPU)* [9], and a model by Sitchinava and Weichert [8], which we refer to as *SWGPU*. Both use a similar architecture.

The models capture a host (CPU) and device (GPU). The device contains unlimited global memory (split into memory blocks) and multiprocessors (MP). Each MP contains cores and shared memory, which is split into memory banks. The global memory can be accessed by the host and by all cores. The shared memory is accessed only by cores on the MP.

The GPU algorithm runs in parallel on the cores of MPs; cores within an MP all perform the same instructions at the same time (in *lockstep*), modelling the concept of a warp. Global memory requests transfer an entire memory block between global memory and shared memory. If requested addresses are within the same memory block, this completes as a single operation, otherwise, multiple operations are required. Shared memory requests complete in constant time provided requested addresses are in distinct banks. Shared memory requests are assumed to be bank conflict free, as bank conflicts are difficult to analyse. The MP waits until all memory requests by the cores have been resolved before proceeding to the next instruction.

The SWGPU models execution in rounds, delimited by synchronisation with the host. Analysis of algorithms is by a cost function; a function of the amount of operations, the amount of memory requests, and the amount of synchronisation.

AGPU analyses algorithms asymptotically by time, number of memory requests and space used in global and shared memory. The AGPU does not capture synchronisation, and disallows algorithms where shared memory used exceeds capacity. Occupancy is measured as a function of shared memory usage and shared memory capacity. The AGPU gives pseudocode for designing algorithms on the model.

Hong and Kim [4] create a predictive tool which can be built into compilers. The tool predicts kernel latency at compile time by analysing the compute and memory access. Their prediction model is not simple, as many calculations are required. Konstandinidis et al. [6] propose the Quadrant-Split method to predict kernel performance on other GPUs after execution on one GPU. When comparing predicted latency and performance with the observed latency and performance, [4] and [6] report a difference (error) of 5.14% and 25.8%, respectively. Neither [4] nor [6] consider the data transfer between CPU and GPU.

## D. CPU GPU Data Transfer

We now discuss work related to CPU GPU data transfer. There has been attempts to model this using software based tools and a cost function, yet to our knowledge there has been no attempt to capture both the GPU kernel and the data transfer in the same abstract model.

Data transfer between the CPU and GPU can often affect the performance of a GPU program under normal usage. Gregg and Hazelwood [14] demonstrate that data transfer between CPU and GPU can affect reported performance and argue that reporting the GPU speedup should include data transfer. A bottleneck was experienced in [15] transferring data between CPU and GPU. Implementing a new data transfer scheme improved the performance by 33%.

Several techniques have been proposed to find the best technique for transferring data between CPU and GPU. Fujii et al. [16] identify that direct memory access, where the GPU and CPU share a unified address space, offers the best performance for large amounts of data transfer. Van Werkhoven et al. [17] produce an analytical tool modelling the data transfer and predicting the best data transfer technique for a GPU program, as it is not feasible to program and test all possibilities.

Boyer et al. [3] propose a function to predict latency of data transfer operations. Their function lowered the difference between predicted and observed speedup from $255\%$ to $9\%$.

### *E. Scope for an Improved Model*

We observe that the SWGPU and the AGPU model different aspects of kernels; SWGPU models the trend of overall running time, whereas AGPU has focused on design and analysis of individual elements of the kernel. Both are equally important, so there is scope for a more comprehensive model combining all elements. Using GPU as a coprocessor requires data transfer between CPU and GPU and this can lower performance if not properly considered. To our knowledge, this is not captured in any abstract GPU model, though it has been well studied in software based tools.

### *F. Our Contribution*

Our contribution consists of an abstract model, called Abstract Transferring GPU (ATGPU), which is an extension of previous models. We introduce new components to capture data transfer between CPU and GPU.

We extend the SWGPU and AGPU architecture, introducing a size constraint on global memory, making the model more realistic. We extend the pseudocode of AGPU to capture data transfer, and we extend the SWGPU cost function to model data transfer and to simulate the cost on a particular GPU. To our knowledge, ATGPU is the first abstract model with this comprehensive array of analysis and design capabilities, and the first abstract GPU model to capture data transfer. A comparison of models is provided in Table I.

| **Item** | **AGPU** [9] | **SWGPU** [8] | **ATGPU** |
|---|---|---|---|
| Pseudocode | ✓ | ✗ | ✓ |
| Time Complexity | ✓ | ✓ | ✓ |
| I/O Complexity | ✓ | ✓ | ✓ |
| Space Complexity | ✓ | ✗ | ✓ |
| Shared Memory Limit | ✓ | ✗ | ✓ |
| Synchronisation | ✗ | ✓ | ✓ |
| Cost Function | ✗ | ✓ | ✓ |
| Global Memory Limit | ✗ | ✗ | ✓ |
| Host/ Device Data Transfer | ✗ | ✗ | ✓ |

TABLE I: Comparison of GPU abstract models

We demonstrate the use of ATGPU and evaluate several example computational problems using the model. We show via experiments that existing models are not able to sufficiently model the actual running time in all cases, as they do not capture data transfer. We show that by capturing data transfer using our model, we are able to obtain more accurate predictions of the actual running time.

The remainder of the paper is organised as follows: Section II discusses our model, and Section III describes how algorithms are analysed in our model. Section IV analyses computational problems using our model, and Section V gives concluding remarks and outlook on future work.

## II. Our Model

We now describe the architecture, execution, and usage of the ATGPU model.

**Architecture.** The architecture of ATGPU is similar to SWGPU and AGPU, with an additional constraint on global memory size. The model captures a *host* (CPU) and a *device* (GPU). Let $ATGPU(p, b, M, G)$ be an instance of the model with $p$ cores in total, $b$ cores and shared memory of $M$ words per MP, and global memory of $G$ words.

Let $MP = \{mp_1, mp_2, ..., mp_k\}$ be the set of MP, therefore $k = \frac{p}{b}$. Let $C_i = \{c_{i,1}, ..., c_{i,b}\}$ be the set of cores of $mp_i \in MP$. All $c_{i,j} \in C_i$ execute the same set of instructions in lockstep. The shared memory of each $mp_i \in MP$ is split into $b$ memory banks, such that $b$ successive words reside in distinct banks. Only $C_i$ can access the shared memory of $mp_i$. The global memory is divided into memory blocks of $b$ words. The host and all $mp_i \in MP$ can access global memory.

**Execution of Algorithms on the Model.** ATGPU executes algorithms in *rounds*, similar to SWGPU. A round begins by the host transferring data to the device global memory. The kernel is then ran on all (or a subset of) $mp_i \in MP$, and on all cores $c_{i,j} \in C_i$. Instructions are executed on $C_i$ in lockstep. If execution paths diverge, all paths are executed. Data must be moved from global memory to shared memory in order for $C_i$ to access it. Upon a memory access instruction, $C_i$ waits until all cores have their memory request resolved.

In a global memory access instruction, if $C_i$ requests words within the same memory block, instructions *coalesce* and complete as a single transaction. If requested words are in $l$ separate memory blocks, $l$ separate transactions occur.

In a shared memory access instruction, if all $c_{i,j} \in C_i$ access words in distinct memory banks, the request completes in constant time. If there is access to words in the same memory bank, a *bank conflict* occurs and requests are serialised. We assume that bank conflicts do not occur, as these are difficult to analyse.

The round ends with output data being transferred from global memory to the host. Synchronisation operations occur, and the subsequent round commences.

**Notation for Pseudocode.** We extend the pseudocode from the AGPU model, allowing for explicit data transfer. Each GPU kernel is placed inside a *wrapper loop*, to execute the instructions on $MP$. If instructions are to be run on only a subset of $MP$, then this is specified within the wrapper loop.

**Pseudocode** Wrapper Loop

```
for all $mp_\rho \in MP[mp_0, ..., mp_{k-1}]$ in parallel do
    for all $c_{\rho,\epsilon} \in C_\rho$ in parallel do
        Instructions ....
    end for
end for
```

In our model, any primitive data type, with vectors and arrays thereof, are allowed as variables. Our model defines

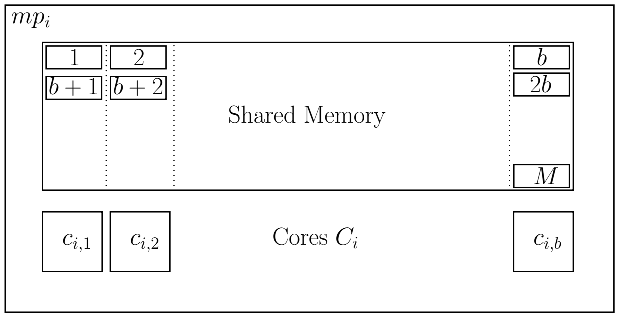


Fig. 1: The multiprocessor of our model. Note the $b$ banks (shown as columns in the shared memory) and the $b$ cores.

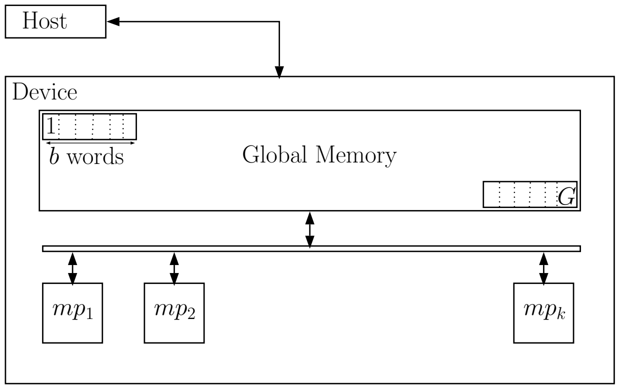


Fig. 2: The device view of our model, with the $k = \frac{p}{b}$ multiprocessors, and the global memory of size $G$, in blocks of $b$ words

three types of variable scope. *Host* variables reside in host memory, only accessible to the host, and their names begin with capital letter. *Global* variables reside in global memory, accessible by the host and all MPs, and their names begin with lower case letter. *Shared* variables reside in shared memory, accessible only by $C_i$ for shared variable on $mp_i$, and their names begin with underscore. Memory access syntax is <*destination*><*operator*><*source*>. Data transfer between host and device uses the $\Lleftarrow$ operator, global memory access uses the $\Leftarrow$ operator, and shared memory access uses the $\leftarrow$ operator. The *if-statement* allows only a single conditional block, in order to reduce diverging execution paths.

## III. Analysing Algorithms on ATGPU

Our model defines the metrics below for an algorithm running on ATGPU. Asymptotic complexity can be measured both on a per-round basis and across the entire algorithm.

**Number of Rounds $R$.** The number of rounds $R$ in the program gives how many rounds are required. As data transfer and synchronisation take a non trivial amount of time, we look to minimise this value. This is from SWGPU.

**Time $t_i$.** The maximum number of operations across all MPs executed in round $i$. This is from both AGPU and SWGPU.

**I/O $q_i$.** The total number of global memory blocks accessed in the round, by all MP. This is from both AGPU and SWGPU.

**Global Memory Space Used.** The maximum number of words stored in global memory across all rounds. If this is greater than $G$, the algorithm cannot be run on our model.

**Shared Memory Space Used.** The maximum number of words stored in shared memory across all MPs and all rounds. If this is greater than $M$, the algorithm cannot be run on our model.

**Data Transfer.** We introduce analysis of data transfer to the model. For round $i$, let $I_i$ ($O_i$ resp.) be the number of words transferred from the host to device (device to host resp.) at the start (end resp.) of the round, referred to as *inward* (*outward* resp.) transfer. The total amount of words transferred between the host and device for all rounds can be measured as: $\sum_{i=1}^{R}(I_i + O_i)$.

**Cost Functions.** The cost function is adapted from the SWGPU, with modifications of data transfer, operation rate, and a GPU-cost.

*Operation Rate* $\gamma$. The cost for a multiprocessor to execute a single instruction is represented by the variable $\gamma$. We see that this corresponds to the clock rate of the GPU. The operation rate $\gamma$ can be set to a value corresponding to a particular GPU for calculating the cost.

*Global Memory Latency* $\lambda$. The cost to access a memory block in global memory is non-trivial; accessing shared memory, when no bank conflicts occur can take 4 cycles, whereas global memory can take in the region of $400 - 800$ cycles. We denote by $\lambda$ this cost, being the number of cycles to access a global memory block.

*Fixed Synchronisation Cost* $\sigma$. The fixed cost synchronisation tasks that need to take place, such as resetting the device, de-allocating and reallocating of data structures, clearing queues, etc. take a non-trivial amount of time. This is represented by $\sigma$.

*Host Device Data Transfer.* Boyer et al. [3] gave a function to determine the time of data transfer between CPU and GPU. We use this to assign cost to data transfer stages. Let $\alpha$ represent the initial overhead of a data transfer transaction, $\beta$ represent the cost of sending a word, and $\hat{I}_i$ ($\hat{O}_i$ resp.) represent the number of data transfer transaction of inward (outward resp.) transfer in round $i$. The function $T_I(i)$ gives the cost of inward data transfer for round $i$: $T_I(i) = \hat{I}_i\alpha + I_i\beta$. Likewise, the function $T_O(i)$ gives the cost outward data transfer for round $i$: $T_O(i) = \hat{O}_i\alpha + O_i\beta$.

*Cost Function.* We say that the cost of an algorithm is upper bounded by Expression (1):

$$\sum_{i=1}^{R}(T_I(i) + \frac{t_i + \lambda q_i}{\gamma} + T_O(i) + \sigma) \ . \tag{1}$$

*GPU-Cost Function.* Expression (1) gives the cost as ran on a "perfect GPU" — a GPU with sufficient resources to concurrently run every thread block in the algorithm. This is an impossible machine, with an unlimited amount of multiprocessors. Like how the AGPU allows a $k$ multiprocessor machine to simulate $w > k$ multiprocessors, we can alter the ATGPU cost function so that it simulates a GPU with $k' < k$ Multiprocessors. Each streaming multiprocessor on a GPU can accommodate $\ell = \min(\lfloor \frac{M}{m} \rfloor, H)$ blocks concurrently, where $H$ represents a hardware imposed limit. The GPU cost function is given as shown in Expression (2), which captures the concept of occupancy:

$$\sum_{i=1}^{R}(T_I(i) + \frac{\lceil \frac{k}{k'\ell} \rceil t_i + \lambda q_i}{\gamma} + T_O(i) + \sigma) \ . \qquad (2)$$

## IV. Evaluation of Our Model

We evaluate our model using three example computational problems, namely, vector addition, reduction, and matrix multiplication. These algorithms have been well studied in the past, and we use our model to focus on the effect of data transfer on their actual running times. We measure the effect on overall running time when the data transfer is included, comparing to when the data transfer is not included. We scrutinise the effectiveness of our model in capturing data transfer and providing a more accurate prediction of overall running time than the SWGPU, which does not capture data transfer.

To do this, we examine the trends of the SWGPU cost function, the ATGPU cost function, the observed total running time, and the observed kernel running time as the input size increases. We use the GPU cost function of our model as the ATGPU cost, and the GPU cost function of our model minus the data transfer as the SWGPU cost. Our model can be shown as useful in cases where the rate of growth for the ATGPU cost function is closer to the actual running time than the SWGPU cost function.

**Hardware Setting.** All experiments are carried out on a custom built machine with the following specifications: *Ubuntu 16.04 OS, CUDA 8, 16GB RAM, AMD A10 5800k APU, nVidia GTX 650 GPU.*

### *A. Vector Addition*

For two vectors $A = (a_1, a_2, ..., a_n), B = (b_1, b_2, ..., b_n)$, the addition is given as $A + B = (a_1 + b_1, a_2 + b_2, ..., a_n + b_n)$. We implement a simple GPU kernel that adds two Vectors of $n$ ints. An element of the answer vector $c_i$ is independent, making this an embarrassingly parallel problem. We assign $n$ threads, with each thread $i$ calculating the value $c_i = a_i + b_i$.

**ATGPU Analysis.** We give pseudocode and analysis of the Vector Addition GPU kernel on the ATGPU model below. The number of rounds is 1, the parallel time complexity is $O(1)$, the I/O complexity is $O(k)$, the global memory complexity is $O(n)$, the shared memory complexity is $O(b)$, the transfer complexity is $O(\alpha + \beta n)$. The cost is $\alpha 3 + \beta 3n + \frac{10+\lambda 3k}{\gamma} + \sigma$. The GPU-cost is $\alpha 3 + \beta 3n + \lceil \frac{k}{\ell k'} \rceil \frac{10+\lambda 3k}{\gamma} + \sigma$. We plot the GPU-cost function in Figure 3a.

**Pseudocode** Vector Addition

```
Input: Two vectors $A, B$ of length $n$
Output: $C = A + B$
1:  a ⇚ A                                                   ▷ Transfer data to Device
2:  b ⇚ B
3:  for all $mp_i \in MP[mp_0, ..., mp_{k-1}]$ in parallel do      ▷ Start GPU
4:      for all $c_{i,j} \in C_\rho$ in parallel do
5:          $\_a[j] \Leftarrow a[ib + j]$
6:          $\_b[j] \Leftarrow b[ib + j]$                        ▷ Work in shared memory
7:          $\_c[j] \leftarrow \_a[j] + \_b[j]$                    ▷ Output to
8:                                                           Global memory
9:          $c[ib + j] \Leftarrow \_c[j]$
10:     end for
11: end for                                                 ▷ Transfer output to Host
12: $C \Lleftarrow c$
```

**Experimental Setting.** We run the Vector Addition kernel on randomly generated data sets, from $n$ = 1,000,000 $\rightarrow$ 10,000,000 with results shown in Figure 3b.

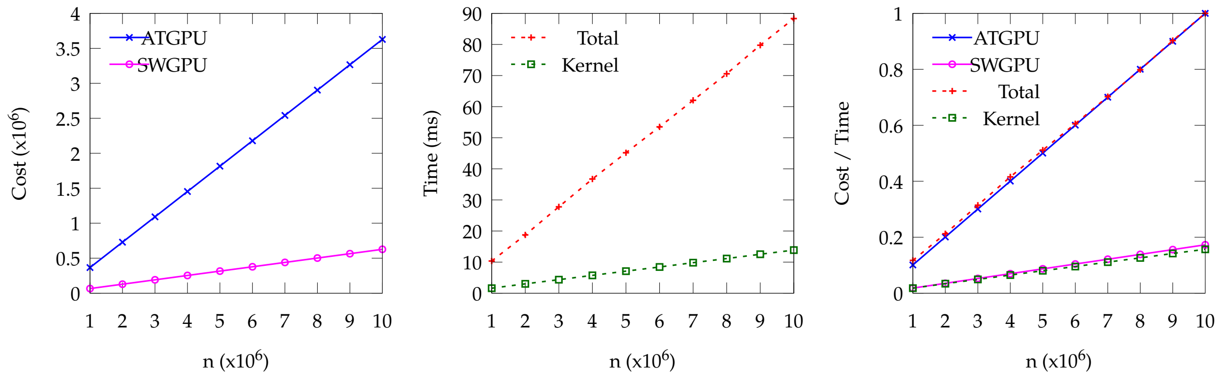
**Discussion.** Figure 3b shows that the growth of total running time is much steeper than the kernel running time, data transfer taking an average of 84% of the total time, meaning data transfer between CPU and GPU has significantly affected the running time of the algorithm. We compare this to Figure 3a, where we see that ATGPU function grows at a much quicker rate than the SWGPU function. In Figure 3c, we have normalised all data on a $0 \rightarrow 1$ scale. We see that the SWGPU function has a much slower rate of growth than the total running time, and that the ATGPU function has a rate of growth which is much closer to the actual total running time. This means that by capturing the data transfer, the ATGPU is able to better predict the total running time of this algorithm, than the SWGPU which does not capture data transfer.

### *B. Reduction*

The reduction of a $n$-sized vector $A$, for some operator $\oplus$, is calculated as $\oplus_{i=1}^{n} a_i$. We implement a simple reduction kernel [18] using the addition operator, to sum an array of $n$ integers, using a tree-based method.

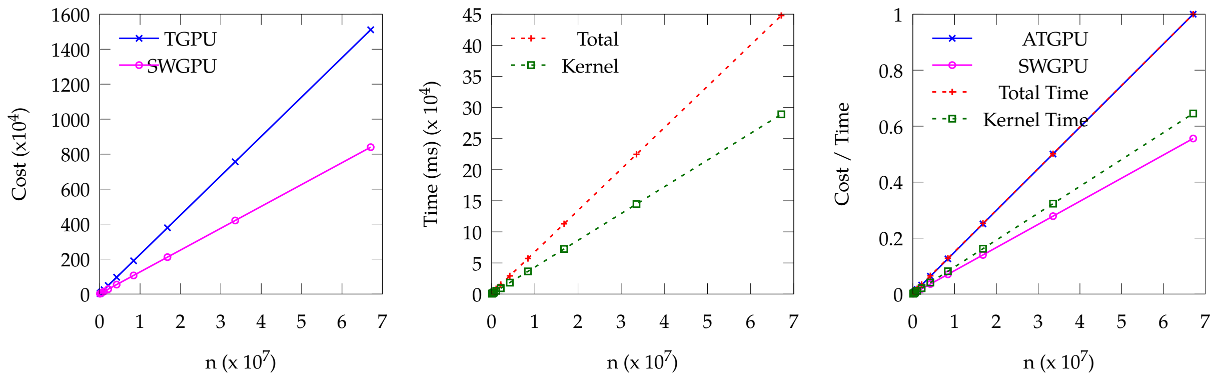
**Pseudocode** Reduction

```
Input: $n$ integers allocated on GPU.
Output: $\sum_{i=1}^{n} A[i]$
1: _A ⇚ A                                   ▷ Transfer input data
2: for $i = 1 \rightarrow \log_b n$ do
3:     Execute Kernel
4: end for
5: Ans ⇚ _A[1]                               ▷ Transfer answer
```



(a) Predicted results. (b) Observed results. (c) Normalised results.

Fig. 3: Results for vector addition.



(a) Predicted results. (b) Observed results. (c) Normalised results.

Fig. 4: Results for reduction.

**ATGPU Analysis.** The algorithm runs as in the “Reduction” pseudocode, each round using the output from the previous round as input.

The number of rounds is $O(\log n)$, the global memory complexity is $O(n)$, the shared memory complexity is $O(b)$, the parallel time complexity is $O(\log b)$, the transfer complexity is $O(\alpha + \beta n)$ and the I/O complexity is $O(\frac{n}{b}\frac{1-\frac{1}{b}^{\log n}}{1-\frac{1}{b}})$. The cost is:

$$O(\alpha + \beta n + \frac{(\log b \log n) + \lambda(\frac{n}{b}\frac{1-\frac{1}{b}^{\log n}}{1-\frac{1}{b}})}{\gamma} + \sigma \log n) \ .$$

The GPU-cost is:

$$O(\alpha + \beta n + \frac{(\lceil\frac{n}{bk'\ell}\rceil\lceil\frac{1-\frac{1}{b}^{\log n}}{1-\frac{1}{b}}\rceil \log b) + \lambda(\frac{n}{b}\frac{1-\frac{1}{b}^{\log n}}{1-\frac{1}{b}})}{\gamma} \\ + \sigma \log n) \ .$$

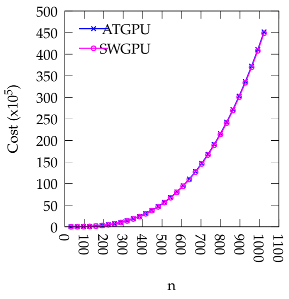
We plot the GPU cost in Figure 4a.

**Experimental Setting.** We run the reduction kernel on randomly generated vectors of 0/1 values, being sizes $n = \{2^{16}, 2^{17}, ..., 2^{26}\}$. We plot the observed results in Figure 4b.

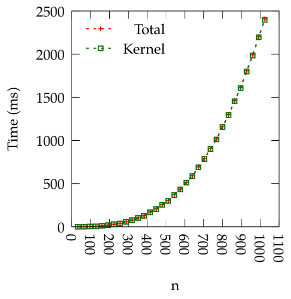
**Discussion.** Figure 4b shows that the growth of total running time is steeper than the kernel running time,though there is not as stark a difference as in vector addition. On average, the data transfer takes 35% of the total running time. We compare this to Figure 4a, where we see that ATGPU function grows at a quicker rate than the SWGPU function. We see in Figure 4c that the ATGPU function has a rate of growth closer than the SWGPU function to the actual total running time. Therefore, as in the vector addition example, capturing the data transfer gives a more accurate prediction of the actual running time.

## *C. Matrix Multiplication*

Finally, we investigate matrix multiplication. For two matrices $A, B$,we multiply them into the matrix $C$. We use a well known GPU method for matrix multiplication in shared memory (introduced in CUDA Programming Guide [2]), modified for the single warp per multiprocessor of our model.



(a) Predicted results.



(b) Observed results.

Fig. 5: Results for matrix multiplication.

**Pseudocode** Matrix Multiplication

**Input:** Two $n \times n$ matrices of integers, $A$ and $B$.
**Output:** $C = A \times B$

```
1: _A ⇚ A                 ▷ Transfer data on Host to Device
2: _B ⇚ B
3: Execute Kernel
4: C ⇚ _C     ▷ Transfer answer data from Device to Host
```

**ATGPU Analysis.** The number of rounds is 1, the parallel time complexity is $O(nb)$ , the parallel I/O complexity is $O((\frac{n}{b})^2(n+b))$ , the global memory used is $O(n^2)$, the shared memory used is $O(b^2)$ , the transfer complexity is $O(\alpha+\beta n^2)$ and the cost is $O(\alpha + n^2\beta + \frac{nb+\lambda\frac{n}{b}^2(n+b)}{\gamma} + \sigma)$. We plot the GPU-cost in Figure 5a.

**Experimental Setting.** We run the matrix multiplication kernel on randomly generated square matrices of side length $n = \{32, 64, ...., 1024\}$. We plot the observed results in Figure 5b.

**Discussion.** We can see from Figure 5b that there is little difference between the kernel running time and the total running time. This means that the data transfer does not affect the running time of this algorithm. This is reflected in the model analysis, but our model is not useful in this case.

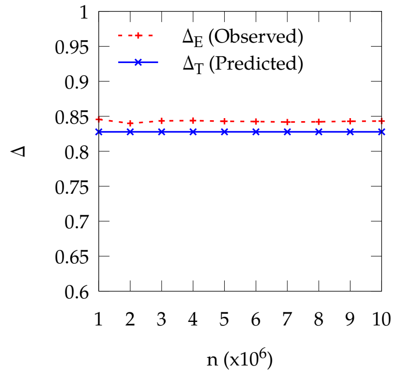
### *D. Summary*

In the three computational problems studied, we see that for vector addition and reduction, it is not sufficient to simply capture the kernel execution for predicting the actual running time. We show that by capturing the data transfer in addition to the kernel, it is possible to obtain a trend that is much closer to the actual running time, than if the data transfer was not captured. We also show a case where our model is not useful; in matrix multiplication, there is little difference between the kernel and total running times, so the kernel can provide an accurate prediction of the total running time in this case.

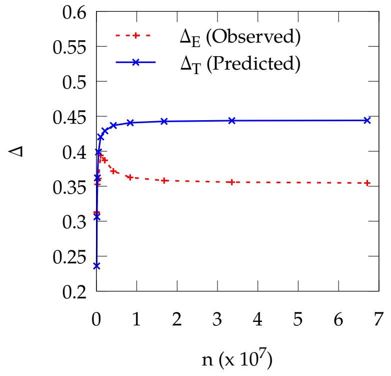
To demonstrate the accuracy of our model, we have also calculated the relative proportions of time/cost allocated to data transfer, and we see that our model has a good level of accuracy, as seen in Figure 6. We see that the predicted proportions of cost allocated to data transfer are on average to within 1.5% of observed proportions for vector addition, to within an average of 0.76% for matrix multiplication, and to within an average of 5.49% for reduction. We also calculate that the SWGPU captures on average only 16% of the actual running time for the vector addition example, and only 58% of actual running time for the reduction example, with 89% of the actual time being captured in the matrix multiplication example.

## V. Conclusion

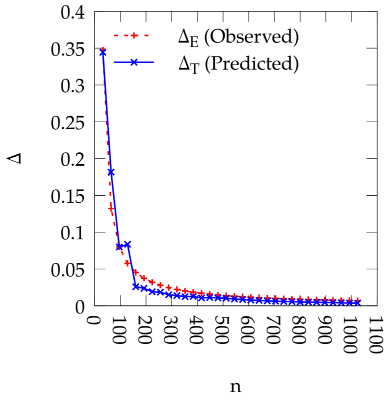
In this paper, we introduce a model called Abstract Transferring GPU (ATGPU), applicable to design and analysis of GPU algorithms. The model is an extension of existing abstract models. ATGPU is, to our knowledge, the first GPU abstract model containing data transfer between host and device as an integral part. The model contains an architecture, a pseudocode and cost functions, allowing an algorithm to be analysed on a "perfect GPU" and simulated on a real GPU. We show via experiments that existing models cannot sufficiently capture all of the actual running time of a GPU algorithm in all cases, as they do not capture data transfer. We show that by capturing data transfer with our model, we are able to obtain more accurate predictions of the GPU algorithm actual running time. We demonstrate two cases where capturing both the kernel and data transfer in our model is useful for better predicting the actual running time, and one case where capturing only the kernel running time is sufficient. The immediate future work is to carry out further experiments on other computational problems to verify our model. Furthermore, it is desirable to verify the model using other GPUs.



(a) Vector addition.



(b) Reduction.



(c) Matrix multiplication.

Fig. 6: Proportions ($\Delta$) of time/cost for data transfer.

## References

[1] "The OpenCL Specification Version 1.2," 2012. [Online]. Available: http://www.khronos.org/registry/cl/specs/opencl-1.2.pdf

[2] nVidia Inc, "Programming Guide:: CUDA Toolkit Documentation," aug 2017. [Online]. Available: http://docs.nvidia.com/cuda/cuda-c-programming-guide/

[3] M. Boyer, J. Meng, and K. Kumaran, "Improving GPU performance prediction with data transfer modeling," in *IPDPSW '13*, pp. 1097–1106.

[4] S. Hong and H. Kim, "An analytical model for a GPU architecture with memory-level and thread-level parallelism awareness," in *SIGARCH Comput. Archit. News*, vol. 37, 2009, pp. 152–163.

[5] K. Kothapalli, R. Mukherjee, M. S. Rehman, S. Patidar, P. J. Narayanan, K. Srinathan, M. Suhail Rehman, S. Patidar, P. J. Narayanan, and K. Srinathan, "A performance prediction model for the CUDA GPGPU platform," in *HiPC '09*, pp. 463–472.

[6] E. Konstantinidis and Y. Cotronis, "A practical performance model for compute and memory bound GPU kernels," in *PDP '15*.

[7] Y. Zhang and J. D. Owens, "A quantitative performance analysis model for GPU architectures," in *HPCA '11*, pp. 382–393.

[8] N. Sitchinava and V. Weichert, "Provably efficient gpu algorithms," *arXiv Prepr. arXiv1306.5076*, 2013.

[9] A. Koike and K. Sadakane, "A Novel Computational Model for GPUs with Application to IO Optimal Sorting Algorithms," in *IPDPSW '14*, pp. 614–623.

[10] S. Fortune and J. Wyllie, "Parallelism in random access machines," in *STOC '78*, pp. 114–118.

[11] L. G. Valiant, "A bridging model for parallel computation," *Commun. ACM*, vol. 33, no. 8, pp. 103–111, 1990.

[12] A. A. Tiskin, "The bulk-synchronous parallel random access machine," *TCS*, vol. 196, no. 1, pp. 109–130, 1998.

[13] L. Arge, M. T. Goodrich, M. Nelson, and N. Sitchinava, "Fundamental parallel algorithms for private-cache chip multiprocessors," in *SPAA '08*. ACM, pp. 197–206.

[14] C. Gregg and K. Hazelwood, "Where is the data? Why you cannot debate CPU vs. GPU performance without the answer," in *IEEE ISPASS*, apr 2011, pp. 134–144.

[15] S. Kato, J. Aumiller, and S. Brandt, "Zero-copy I/O processing for low-latency GPU computing," in *ICCPS '13*, pp. 170–178.

[16] Y. Fujii, T. Azumi, N. Nishio, S. Kato, and M. Edahiro, "Data Transfer Matters for GPU Computing," *ICPADS '13*, pp. 275–282.

[17] B. V. Werkhoven, J. Maassen, F. J. Seinstra, and H. E. Bal, "Performance models for CPU-GPU data transfers," in *Proc. - 14th IEEE/ACM CCGrid 2014*, pp. 11–20.

[18] M. Harris, "Optimizing parallel reduction in cuda," 2008. [Online]. Available: http://developer.download.nvidia.com/assets/cuda/files/reduction.pdf