Determining the difficulty of accelerating problems on a GPU

Dale Tristram, Karen Bradshaw

Department of Computer Science, Rhodes University, P. O. Box 94, Grahamstown, South Africa

ABSTRACT

General-purpose computation on graphics processing units (GPGPU) has great potential to accelerate many scientific models and algorithms. However, since some problems are considerably more difficult to accelerate than others, ascertaining the effort required to accelerate a particular problem is challenging. Through the acceleration of three typical scientific problems, seven problem attributes have been identified to assist in the evaluation of the difficulty of accelerating a problem on a GPU. These attributes are inherent parallelism, branch divergence, problem size, required computational parallelism, memory access pattern regularity, data transfer overhead, and thread cooperation. Using these attributes as difficulty indicators, an initial problem difficulty classification framework has been created that aids in evaluating GPU acceleration difficulty. The difficulty estimates obtained by applying the classification framework to the three case studies correlate well with the actual effort expended in accelerating each problem.

KEYWORDS: GPGPU, OpenCL, problem difficulty classification

CATEGORIES: D.1.3, D.2.8

1 INTRODUCTION

Accelerating scientific problems on graphics processing units (GPUs) can result in orders of magnitude speedup over CPU-based solutions.

With Cray's Titan, the world's second fastest supercomputer[1], making extensive use of GPUs, more scientists are likely to be interested in using these devices to accelerate their models and algorithms. However, because of the way in which GPUs have been designed, some problems are considerably harder to accelerate than others.

For scientists unfamiliar with the architecture and programming of GPUs, the distinction between easily accelerated problems and those that are very difficult (yet possible) to accelerate is likely to be unclear. For this reason, novice GPU programmers may be discouraged if the performance achieved does not meet their expectations.

One way of addressing this would be to create a problem difficulty classification system that could provide users with information on the level of problem difficulty and the kind of knowledge and optimisations necessary to achieve satisfactory speedup on a GPU.

However, in order to create such a system, we first need to identify the problem attributes that are important in distinguishing the different levels of difficulty. This paper sets out to determine some of these attributes through the acceleration of three different problems and then validate them by applying the classification framework to the problems accelerated.

Email: Dale Tristram `d.tristram@boost.za.net`, Karen Bradshaw `k.bradshaw@ru.ac.za`

[1]http://www.top500.org/lists/2013/06/

Section 2 provides a brief overview of GPU computing. Sections 3, 4, and 5 detail the acceleration of a hydrological model, k-difference string matching, and a radix sort, respectively. Section 6 discusses the creation of the classification framework and its application to the accelerated problems. Finally, Section 7 concludes.

2 GPU COMPUTING

There are a few key differences between the architectures of common GPUs and CPUs that must be understood in the context of general-purpose computation on graphics processing units (GPGPU).

A brief overview of the processing and memory models of a GPU is presented for some insight into these differences, as well as a high-level overview of the GPU computing framework used in this study.

2.1 Graphics Processing Units

Although a number of different GPU architectures exist, modern GPUs all share certain architectural similarities [1]. The AMD Radeon HD7970, hereafter referred to as the HD7970, is used as the reference GPU when explaining the general GPU processing and memory model.

2.1.1 GPU Processing Model

One of the fundamental differences between GPUs and CPUs is the kind of processing that is prioritised, and consequently their respective number of processing units. Modern CPUs typically have between two and eight cores, and have been designed to maximise the speed of single threads of execution on those cores [2]. Conversely, GPUs have been designed to maximise the total throughput of many threads of execution, and have hundreds to thousands of stream processors distributed over a number of compute units [2, 3, 4]. For instance, the HD7970 has 32 compute units, each containing 64 stream processors, giving a total of 2048 stream processors [5]. GPUs are able to achieve a high stream processor density by effective use of single-instruction, multiple data (SIMD) processing, where many processing elements are packed together, sharing hardware resources at the cost of independent instruction execution [5, 3]. As a whole, GPUs are single program, multiple data (SPMD) devices since each compute unit is independent. Modern CPUs, which are multiple instruction, multiple data devices, also support SIMD processing through the use of extensions such as streaming SIMD extensions and advanced vector extensions, but on a much smaller scale than modern GPUs [3].

2.1.2 GPU Memory Model

In contrast to CPU memory, GPU memory has been designed to maximise bandwidth output rather than minimise access latency [3]. However, with sufficient parallelism, the high memory access latency can be mostly hidden by doing alternative computation. The five kinds of GPU memory usually available to the programmer are: private memory, local memory, global memory, constant memory, and image/texture memory [1]. These memories and their relationships are illustrated in Figure 1 for AMD's Southern Island GPUs. The movement of data between these five memory banks is left entirely to the programmer, making it crucial for the programmer to understand the attributes of each of the available GPU memories to make good choices on where to store different program data. This is unlike CPU architectures, where data caching is mostly hidden from the programmer. Transferring data to GPU memory is done through the PCI Express bus for discrete GPUs. Since the PCI Express bus is considerably slower than the GPU memory, and possibly the host memory too[2], it can be a significant performance bottleneck for bandwidth intensive applications.

2.2 OpenCL

OpenCL is an open, royalty-free standard developed by the Khronos Group aimed at providing a single platform for parallel computation across heterogeneous computation devices [6]. OpenCL was used to accelerate the problems discussed in this paper, and thus we give a high-level overview of its execution model and programming model.

The following information on OpenCL was sourced from the official OpenCL 1.2 specification [6], unless referenced otherwise.

[2]Standard computer memory is faster than PCI Express 2.x, but not faster than the less widely used PCI Express 3.x, unless the memory is operating in dual channel mode.

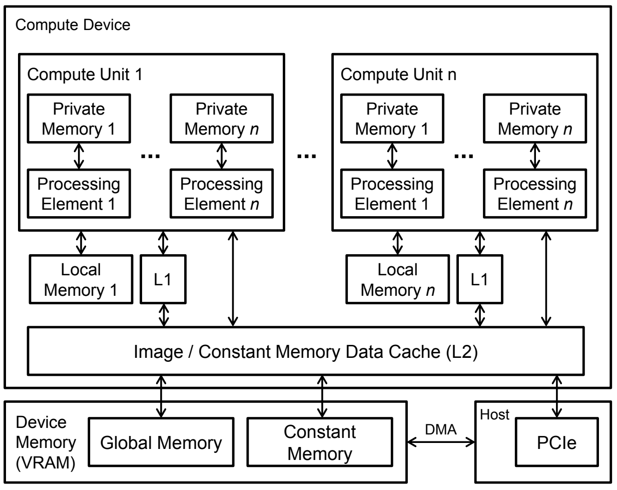


Figure 1: Memory hierarchy and interrelationships of Southern Island devices, adapted from [5].

2.2.1 Execution Model

OpenCL code is executed by enqueuing *kernels* on an OpenCL accelerator (any device with an OpenCL driver), which are functions that act as entry points into the OpenCL program. To schedule a kernel for execution, the kernel's *NDRange* must be specified. An NDRange is simply an index range that can have between one and three dimensions. Each index in the NDRange corresponds to a unique thread of execution to be scheduled on the OpenCL accelerator. OpenCL divides the index space into groups known as *work-groups*, which are groups of *work-items* guaranteed to be executed together on the same compute unit, allowing them to share data.

On AMD devices, work-groups are further divided into a series of *wavefronts* of at most 64 threads. An illustration of an OpenCL NDRange is provided in Figure 2. This division of work-items into groups allows for divergent code to be executed efficiently by different work-groups, otherwise known as SPMD execution. A kernel is given access to data by specifying OpenCL buffers, images, or image arrays as kernel arguments. When a kernel is run, each OpenCL thread executes exactly the same kernel program, with the only difference being its thread index, allowing it to select different input data.

2.2.2 Programming Model

The language used for writing OpenCL programs is a variation of the C99 specification, with added extensions for parallelism. OpenCL programs can be written in a data-parallel style, task-parallel style, or a combination of the two. The data-parallel style, which is the most commonly used approach, expresses parallelism by executing the same code on multiple threads with different input data. In the task-parallel style, parallelism is expressed by running different "tasks", or OpenCL kernels, in parallel, and using vector data types.

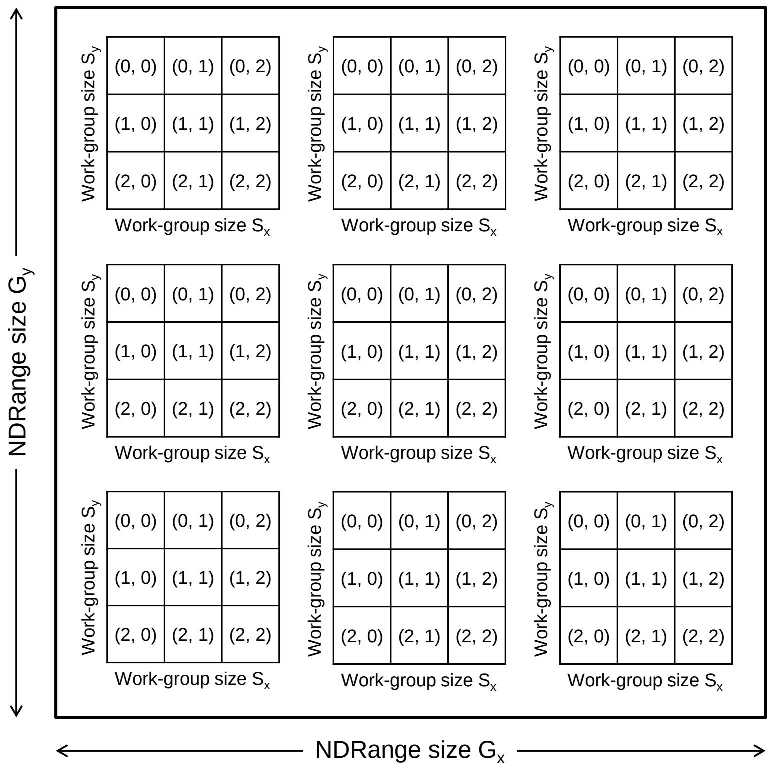


Figure 2: Illustration of a 2-dimensional NDRange composed of work-groups containing work-items (adapted from [6]).

3 HYDROLOGICAL MODEL

Hydrological models are simplified representations of certain processes within the hydrological cycle. These models are primarily used to increase our understanding of the observed processes and to make hydrological predictions or estimations [7]. A recent trend in hydrological modelling is the use of uncertainty analysis [8]. Using this approach, a model is run thousands of times using different input parameters. This can take a very long time on a CPU, but could stand to benefit greatly from GPU acceleration owing to the problem's SIMD-like nature. The hydrological uncertainty model accelerated here is based on an adapted version of the Pitman rainfall-runoff model used for water resource estimation.

3.1 The Pitman Model

The Pitman model is a conceptual type, monthly time-step, semi-distributed (sub-catchment) model that includes some 23 parameters that govern the algorithms defining the hydrological storages and processes such as evapotranspiration, interception, surface runoff, soil moisture storage, interflow, groundwater recharge and drainage, and catchment routing. An overview of the hydrological processes and their relationships for this version of the model is illustrated in Figure 3. The ability of the model to accurately represent the hydrological response of any given catchment is reliant on the correct specification of the model parameters. The uncertainty version of this model is designed to assist in the estimation of these parameters, and allows the model results for many different options within the feasible parameter space to be explored [8]. This is done by running many ensembles (or instances) of the model, each with a different set of parameters. Tens of thousands of ensembles are desirable to adequately explore the parameter space, but this takes hours to complete on a modern desktop CPU, thus making GPUs an attractive alternative.

3.2 Implementation and Optimisations

The core model is approximately 1550 lines of Delphi code. Rather than extracting this code from the Delphi program and attempting a direct conversion into OpenCL, the model was first implemented in an alternative CPU language with good debugging tools to potentially save some debugging on the GPU, which is more difficult. C# was selected as the alternative language because of its syntactical similarity to OpenCL, and the high quality debugging tools available for it. Once the model had been successfully implemented in C#, it was converted into OpenCL. The overall process followed is illustrated in Figure 4.

This approach proved to be fruitful, as conversion errors were identified in the C# version that may have taken significantly longer to detect if a straight conversion into OpenCL was attempted because of inferior debugging tools and the scale of parallelism.

A direct conversion of a model or algorithm into OpenCL rarely results in optimal use of the GPU. This conversion was no exception, and we had to implement one significant optimisation to get a satisfactory speedup. This optimisation involved changing the

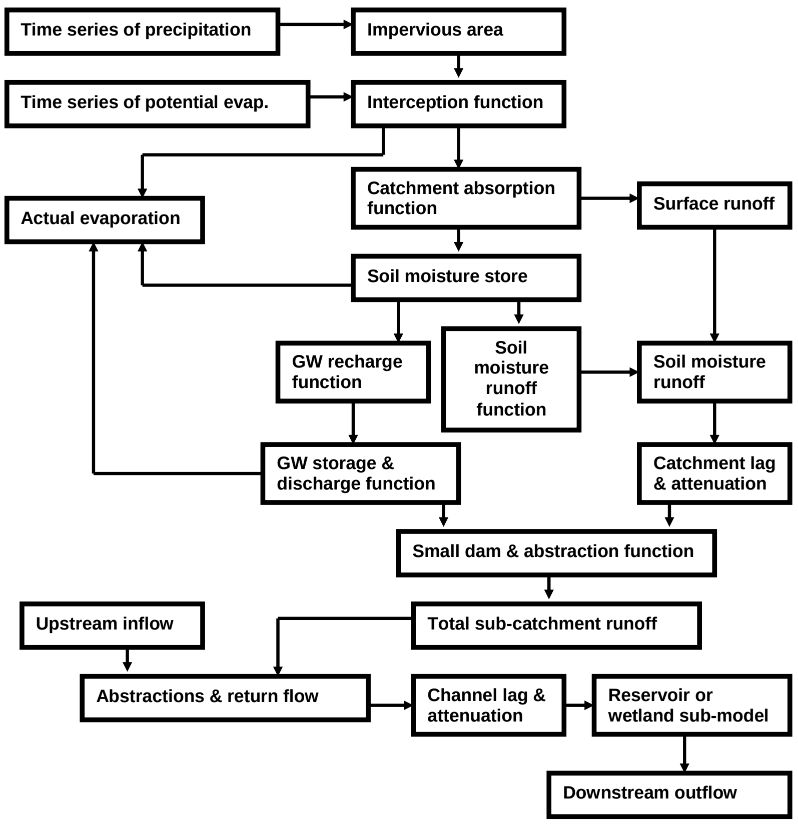

Time series of precipitation
Impervious area
Time series of potential evap.
Interception function
Actual evaporation
Catchment absorption function
Surface runoff
Soil moisture store
GW recharge function
Soil moisture runoff function
Soil moisture runoff
GW storage & discharge function
Catchment lag & attenuation
Small dam & abstraction function
Upstream inflow
Total sub-catchment runoff
Abstractions & return flow
Channel lag & attenuation
Reservoir or wetland sub-model
Downstream outflow

Figure 3: Conceptual process diagram of the Hughes et al. [8] version of the Pitman model.

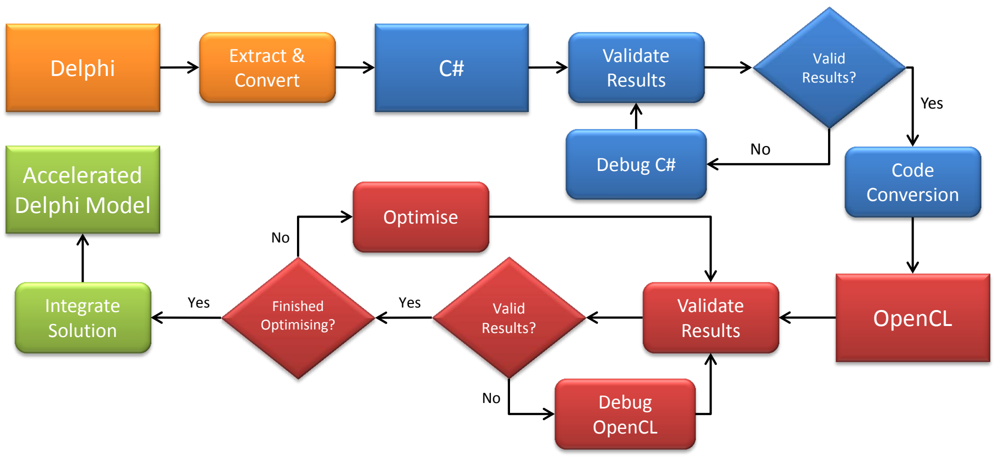


Figure 4: Approach taken to accelerate the uncertainty version of the adapted Pitman rainfall-runoff model.

memory access pattern, or memory stride, which is relatively simple to understand and implement, and can be applied to most GPU acceleration problems.

Memory Stride

A memory stride is defined by AMD's Accelerated Parallel Processing guide as "the increment in memory address, measured in elements, between successive elements fetched or stored by consecutive work-items in a kernel" [5]. Optimising the memory stride can avoid or reduce GPU memory bank conflicts [5], which serialise memory requests. A one-unit memory stride is usually suitable for GPU programming, and is a means of avoiding channel conflicts on the Southern Island architecture [5], and taking advantage of memory access coalescing on other architectures. The data layout of the original model resulted in an undesirable many-unit stride between data accesses of consecutive work items since the data for each model instance were stored in large data records. To change this into a one-unit stride, the variables within each of the model instance structs were written to memory locations x places apart, where x is the number of model instances. A comparison between the original layout and this new layout is illustrated in Figure 5.

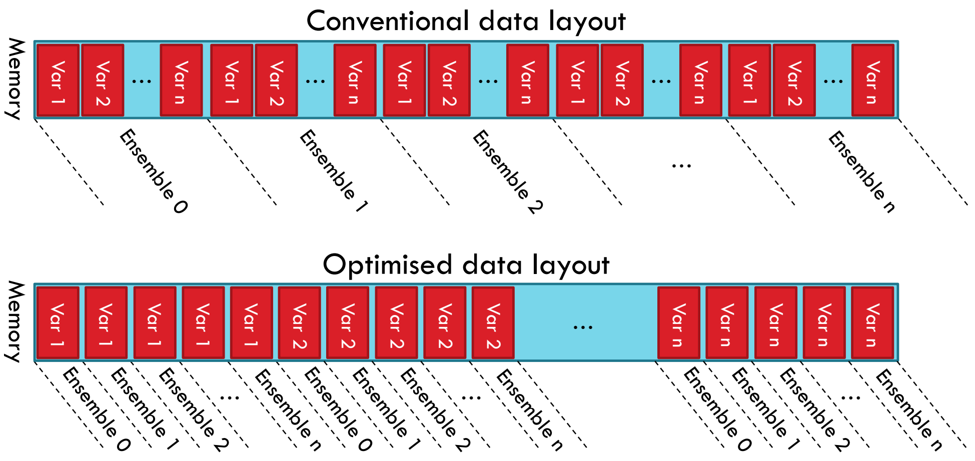


Figure 5: Original data layout in memory compared to the optimised layout.

3.3 Results

The performance of the model was evaluated by averaging the time taken to complete five runs of a number of different model configurations and data. This was done on a computer with an Intel i7 3770 CPU, and an AMD Radeon HD7970 GPU. The results can be seen in Figure 6.

The GPU is just over 12x faster than the multithreaded C# implementation, which is a healthy order of magnitude speedup. To assess the consistency of these results and the impact of running smaller or greater numbers of ensembles, the model was also benchmarked on a second actual dataset (representing the Caledon River basin in South Africa), the results of which are shown in Figure 7.

These results reveal that if 15 000 or more ensembles are run, the GPU's speedup over the CPU implementations is consistent with the results in Figure 6. However, if fewer than 15 000 ensembles are run, the speedup starts declining. This is indicative of the GPU not having sufficient work to hide memory access latency. Considering that running a large number of ensembles is desirable in uncertainty modelling, this should not be a cause for concern. The performance impact of data transfers to and from the GPU was also measured and found to account for less than 0.1% of the total program time, and this value decreased with higher ensemble counts.

We therefore conclude that this model was relatively easy to accelerate as it only required a single GPU optimisation to achieve an order of magnitude speedup.

4 *K*-DIFFERENCE STRING MATCHING

There are many GPU solutions for accelerating approximate string matching, but very few for accelerating large numbers of k-difference calculations, as could be used in malware and spam detection [9, 10]. Although the parallelism of this problem is similar to that of the uncertainty model in Section 3, the high ratio of memory transactions to computation makes it more challenging to obtain a satisfactory speedup.

4.1 *K*-Difference Algorithm

K-difference algorithms calculate the *edit distance* between two strings, which is the minimum number of insertion, deletion, and replacement operations needed to make two strings identical. The k value specifies the maximum number of weighted errors between two strings after which the strings are considered sufficiently different to be classed as non-matching. The basic algorithm uses a dynamic programming approach [11]. A difference matrix $\mathrm{C}_{0..i,0...j}$ is built, where *0..i* represents the characters in the test string m, and *0..j* represents the characters in the input string n. The first row $\mathrm{C}_{0,j}$ is populated with its column index, and the first column $\mathrm{C}_{i,0}$ is populated with its row index. The rest of the table is populated by applying this rule:

```
C_{i,j} = (if $m_i == n_i$):
        $C_{i-1,j-1}$
else:
        $1+\min(C_{i-1,j-1}, C_{i-1,j}, C_{i,j-1})$
```

After the table has been populated, the last cell in the table, $\mathrm{C}_{i-1,j-1}$, contains the difference value between the two strings. This value can then be checked against the threshold k to determine whether the strings are sufficiently similar for the matching algorithm. Computing k-difference problems using this algorithm is slow with a time complexity of $\mathrm{O}(mn)$, but the speed can be significantly improved by using Ukkonen's cut-off heuristic and bit-parallelism. Ukkonen's heuristic ensures that only relevant sections of the dynamic programming matrix are calculated, while bit-parallelism involves packing the columns of the matrix into words (w), such as unsigned integers, and transitioning between columns using bit operations [12, 13]. Even with these improvements, the algorithm still has an average time of $O(kn/w)$ [13]. Although this may be accept-

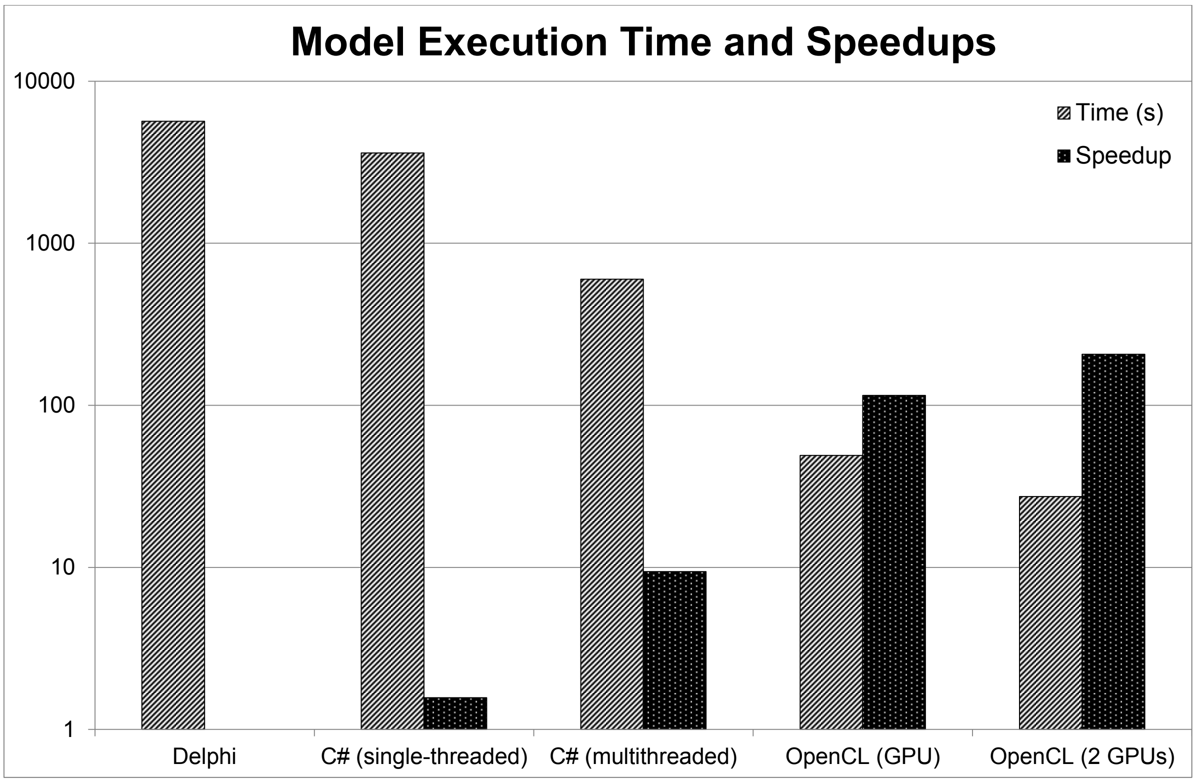


Figure 6: Performance results of different implementations of the hydrological uncertainty model.

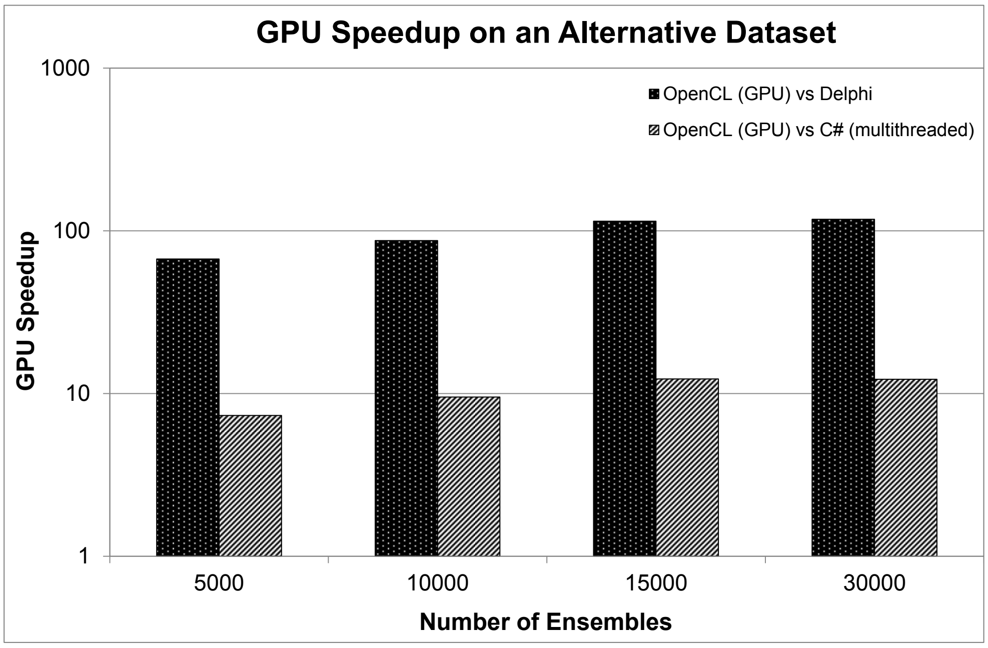


Figure 7: GPU speedup when running 5 000 to 30 000 ensembles of the hydrological model on the Caledon River dataset.

able for applications that perform a limited number of comparisons, it is impractical for applications that continuously perform large numbers of comparisons without massively parallel hardware, such as GPUs. The case we consider is spam or malware detection, where signature test strings are tested against large numbers of input strings.

4.2 Implementation and Optimisations

The core algorithm for computing a single column in the dynamic programming matrix was based on Hyyrö's implementation [14]. Since the algorithm, expressed in approximately 84 lines of code, operates predominantly on private (or register) memory, implementing it in OpenCL was very straightforward. One difference between the CPU and GPU versions of the core algorithm was the GPU version's use of 32-

bit unsigned integers as bit-vectors for storing column information as opposed to the CPU version's use of 64-bit integers. Although using larger words is usually beneficial if the length of the strings compared is greater than the word size, 64-bit memory transactions are not as optimised as 32-bit transactions on our GPU [5], and thus result in worse performance. With the core algorithm being very similar to the CPU version, the challenge of creating a fast GPU version lay in optimising the memory accesses and scheduling.

4.2.1 Memory Loads & Caching

To ensure the string data in global memory were read as fast as possible, the string characters were loaded in batches into temporary private memory. This is faster than reading and processing each character individually since it better utilises the available global memory bandwidth. Another advantage of reading the characters in batches before processing is the use of private memory as a temporary cache to speed up multiple accesses to the same data. This optimisation resulted in an average speedup of 1.8x for short strings and 1.3x for long strings.

4.2.2 Intra-Group Cooperation

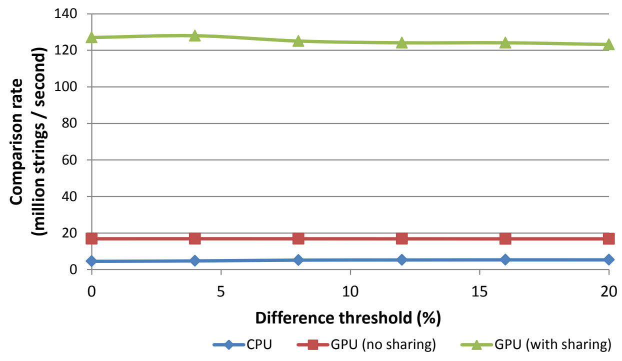
Although each thread calculates its own k-difference problem, one common element between the threads is the test pattern. This was leveraged by using intra-group thread cooperation to cooperatively read the string from global memory and generate the required bit-vector arrays (in the form of unsigned integers). These were then stored in faster local memory to be used by all the threads within the group. Since each thread reads the same test pattern from local memory during the running of the algorithm, local memory loads benefit from broadcasted reads [5]. The speedup from this optimisation ranged from an average of 37x for short strings to 2.9x for long strings.

4.2.3 Kernel Scheduling

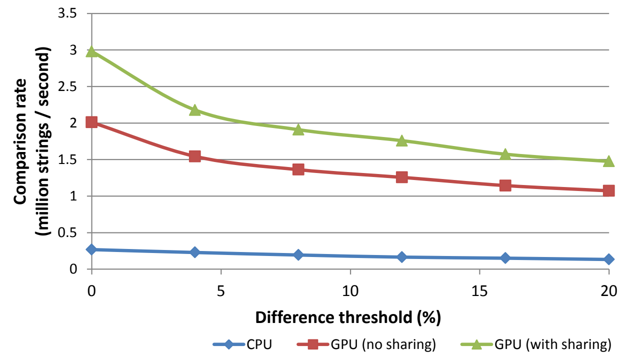
To reduce scheduling overhead, the NDRange for the kernel was specified to be the number of threads that would result in the optimal number of wavefronts per GPU compute unit, as found through kernel profiling. Using this method, each work-item may perform multiple k-difference calculations.

4.3 Results

The GPU implementation was benchmarked against an optimised multithreaded version of Hyyrö's bit-parallel algorithm running on an Intel i7 3770 CPU. Four categories of test data were used: short texts of no more than 64 characters, long texts of between 256 and 480 characters, a small alphabet of four characters, and a large alphabet of 64 characters. For brevity, we show the results of the extreme combinations of the categories only. The number of strings compared was also evaluated, but we found the difference to be negligible (provided the GPU had sufficient work). The results can be seen in Figure 8.



(a) Small alphabet with short patterns.



(b) Large alphabet with long patterns.

Figure 8: Performance of the GPU compared to the CPU for (a) a small alphabet with short patterns, (b) a large alphabet with long patterns, with varying difference thresholds for each.

It is clear from the results that the GPU performs considerably better with shorter patterns. Given that global memory is the GPU's performance bottleneck and shorter patterns require fewer loads from global memory, this is unsurprising. An additional trend can be seen where the GPU's advantage over the CPU declines with an increasing difference threshold for short patterns. This can be attributed to increased pressure on the GPU's global memory, as more comparisons are necessary before the early exit threshold is reached (or a match is found). Overall, the speedup of the GPU solution over the CPU ranges from 9.5x to 28.1x. The performance of the GPU without using the data sharing optimisation is also added for comparison, since this optimisation is what makes this problem harder than the problem accelerated in Section 3. The speedup of the non-data sharing version over the CPU is between 3.1x and 8.5x, a considerable drop from the data sharing version.

The benchmarks were carried out with the GPU program configured to send data both to and from the GPU, which is the least desirable configuration for data transfers. The overall contribution of data transfers to the total runtime of the GPU program was evaluated with the results for various configurations given in Table 1. Interestingly, these results show that

Table 1: Data transfer contribution to the total GPU time of a number of different configurations at a threshold of 4%. ‘Strings’ has been abbreviated to ‘str’.

Predicate strings	500	1000	2000	4000
Small α, short str.	11.7%	9%	8.9%	8.3%
Large α, long str.	0.2%	0.2%	0.2%	0.1%

the data transfer time impacts comparisons between short strings the most with data transfers contributing up to 11.65% of the total GPU time compared to up to 0.16% for long strings. This shows that the compute time increases considerably faster than the data transfer time with longer strings.

5 RADIX SORT

The need for sorting is found in many Computer Science problems. Although highly optimised sorting algorithms have been developed for CPUs, sorting very large datasets such as those found in existing GPU applications can still be a performance bottleneck [15]. Unlike the problems discussed in Sections 3 and 4, sorting does not map easily onto the GPU’s architecture because of the inherent and irregular data dependence between the records to be sorted [15]. However, using the right techniques, a modern GPU can achieve a sizeable speedup over a modern CPU for certain use cases.

5.1 Radix Sort Algorithm

A least significant digit radix sort algorithm works by repeatedly sorting the input keys based on increasingly higher value sections of the physical representation of the keys. The number of sections, and consequently sorting passes, is determined by the width of the section, otherwise known as the radix. A simple serial radix sort of four values is illustrated in Figure 9.

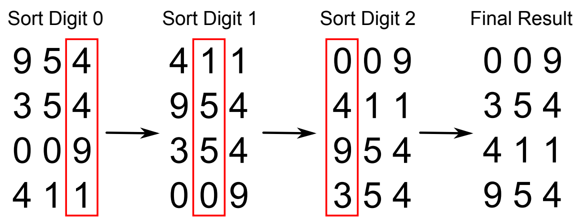


Figure 9: Simplistic illustration of the steps performed in a radix sort. In this example, the radix is a single digit, i.e., units, 10’s and 100’s are sorted in turn.

Implementing an efficient parallel version of this algorithm on the GPU is difficult because of the data dependence properties of the algorithm, and would require an experienced GPGPU developer. The current fastest GPU radix sort algorithm was created by Merrill and Grimshaw and is written in CUDA for NVIDIA GPUs [15]. To understand the complexities of accelerating a problem of this level of difficulty on a GPU, we re-implemented Merrill and Grimshaw’s algorithm in OpenCL for AMD’s Tahiti range of GPUs.

A very high-level overview of the algorithm implementation is illustrated in Figure 10. A more detailed description of the implementation can be found in [15]. The implementation consists of three phases, each of which is implemented in a separate GPU kernel:

Kernel 1: The purpose of this kernel is to determine the aggregate number of keys that fall into the different bit pattern buckets for each of the work-groups, where the keys are decoded based on the current offset in the key and the chosen radix. Each thread reads a number of keys from global memory and increments the appropriate bucket in local memory depending on the key’s decoded value. The decoded bit pattern is simply the relevant section of the value as determined by the radix and current sorting pass. The buckets in local memory are then serially reduced to obtain the final group bucket counts, which are saved in global memory.

Kernel 2: This kernel performs a prefix scan of the bit pattern buckets saved by the first kernel to obtain the global bucket offsets for each of the work-groups. This is done to provide each work-group with an offset in global memory for each bit pattern bucket to which it can scatter its keys.

Kernel 3: The final kernel can be conceptually separated into four phases. In the first phase, the keys are re-read from global memory, decoded, and bucketed according to their bit pattern. This computation is redundant since it was done in Kernel 1, but is repeated because it is faster than saving and loading the results to and from global memory. In the second phase, the threads within the work-groups cooperate to determine their inter-group bucket offsets using prefix scans. The third phase serves to optimise the scattering of the keys by performing an intra-group key exchange that results in the threads holding keys with offsets that would result in ordered writes to global memory. The fourth and final phase adds the global group bucket offsets to the local offsets to get the final global memory offsets, and scatters the keys accordingly.

Since the kernels only operate on a section of the input keys, the results output by Kernel 3 are intermediate and are used as the input for the next iteration. The number of iterations required to fully sort the keys depends on the size of the key and the chosen radix. For a typical key size of 32 bits, eight iterations of the kernels would be needed to fully sort the keys using a radix of four.

5.2 Implementation

Our version of Merrill and Grimshaw’s GPU radix sort algorithm was implemented by rewriting a revision of the CUDA algorithm in OpenCL. Apart from the standard CUDA to OpenCL syntax changes, many of the required changes involved modifying sections of

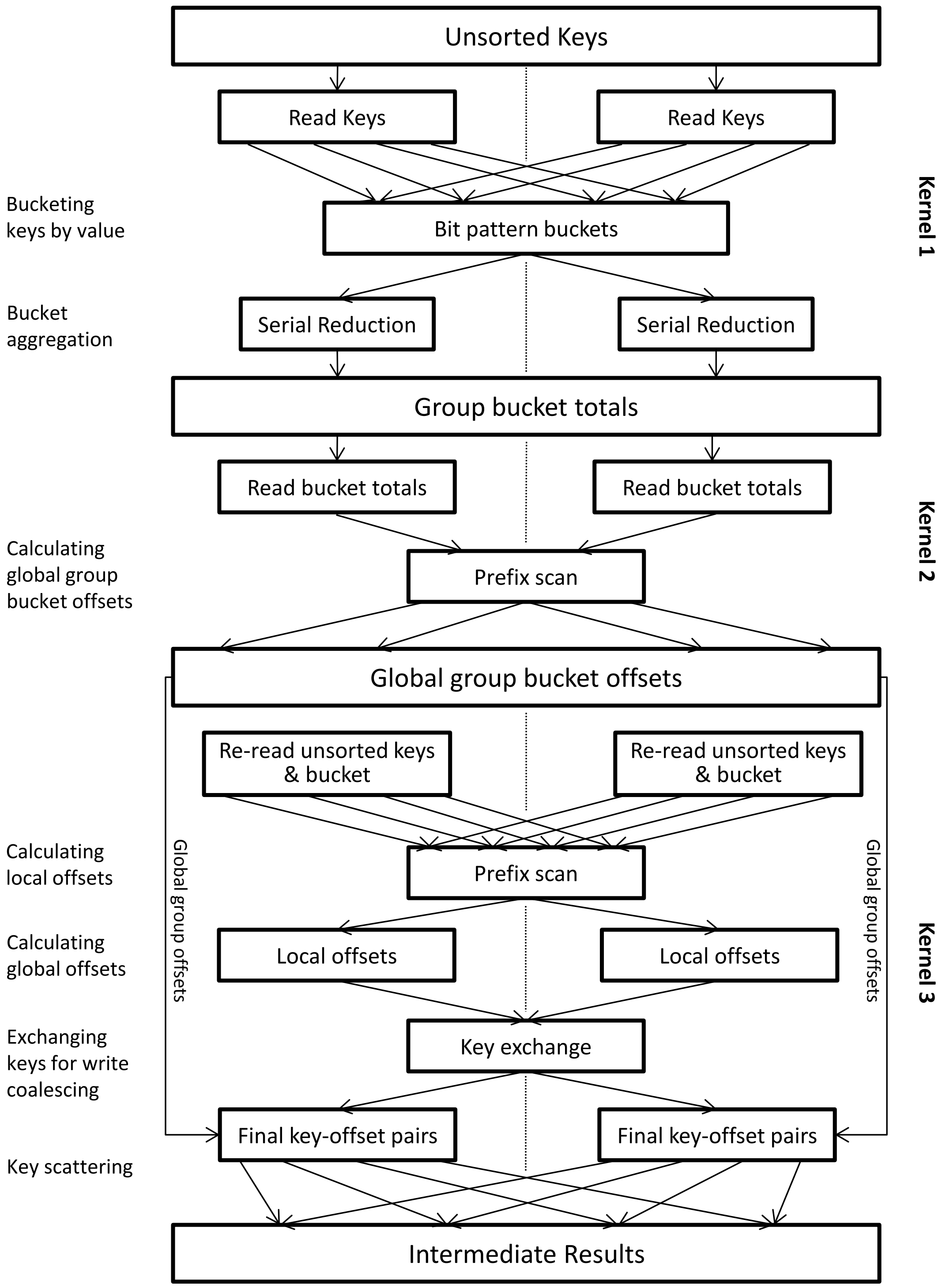


Figure 10: High-level overview of the steps performed in Merrill and Grimshaw's GPU radix sort algorithm. The dotted line in the centre demarcates sections of independent computation.

code to use a wavefront of 64 instead of 32 threads and modifying GPU architecture-specific settings that governed the memory usage pattern of the algorithm. Even though the algorithm was explicitly designed for the GPU, implementing it in OpenCL was still challenging because of the low-level optimisations that needed to be understood and adapted for our particular GPU's architecture.

A number of different strategies and techniques are used in the algorithm to ensure the GPU is utilised as efficiently as possible. We describe a few of them that have been identified as representative of problems of this level of difficulty.

5.2.1 Computational Granularity

Each GPU work-group processes a portion of the total number of keys, which is in the thousands for problem sizes large enough to warrant the use of GPUs. To ensure efficient use of both the stream processors and GPU memory, these keys are processed in batches of a size tied to the target GPU's architecture. Furthermore, throughout the radix sort algorithm the number of memory loads done prior to computation has been set to depend on the target GPU architecture. This is because different GPU architectures have different efficient ratios of computation to memory transactions as a result of different memory and stream processor configurations. It is therefore necessary to optimise the computational granularity based on the target GPU architecture for best performance.

5.2.2 Synchronisation-Free Cooperation

It is common for synchronisation barriers to be used when GPU threads write to local memory to ensure memory consistency is maintained. However, these barriers do incur a performance penalty [5], and are best used only when necessary. Synchronisation-free thread cooperation was used throughout the radix sort algorithm by keeping the number of threads participating in thread cooperation to within a single wavefront. Since wavefronts are executed atomically [5], synchronisation is not needed between the threads.

5.2.3 Loop Unrolling

Loop unrolling (otherwise known as loop unwinding) is the practice of minimising or removing loop control flow logic by directly embedding multiple iterations of the loop into the code. Wherever it was possible, loops were unrolled either through a compiler directive or manually by using a tiered function hierarchy. With the tiered function hierarchy, a particular tier calls lower tier functions until the lowest function is reached, which contains the unrolled code.

5.2.4 Memory Packing

When working with values much smaller than can be held by the value type in local or global memory, packing multiple values into a single word can be an effective way of reducing memory load. This technique is used with local memory repeatedly in the algorithm, where an integer is re-interpreted as four separate char values.

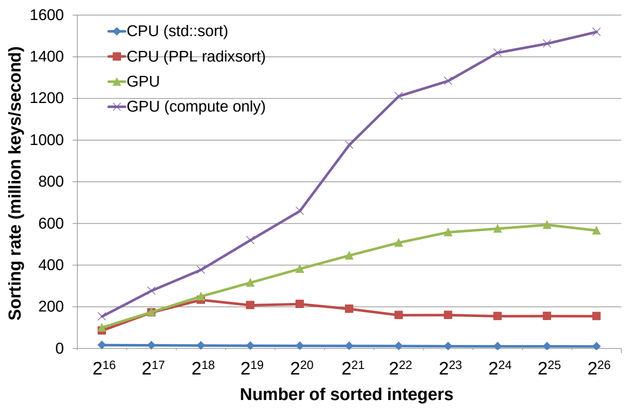
5.2.5 Kernel Fusion

In programs where common operations need to be performed on the GPU data, it may seem sensible to use existing optimised solutions provided by libraries such as Boost.Compute[3] for OpenCL, and the Data-Parallel Primitives Library[4] for CUDA, as has been done in previous solutions [16]. However, there is a significant performance penalty for doing so as all the data must be passed from one kernel instance to another through global memory. This implementation integrates all the required operations into existing kernels, thereby reducing the aggregate memory workload by allowing the results from one step to be passed to the next through local or private memory [15].

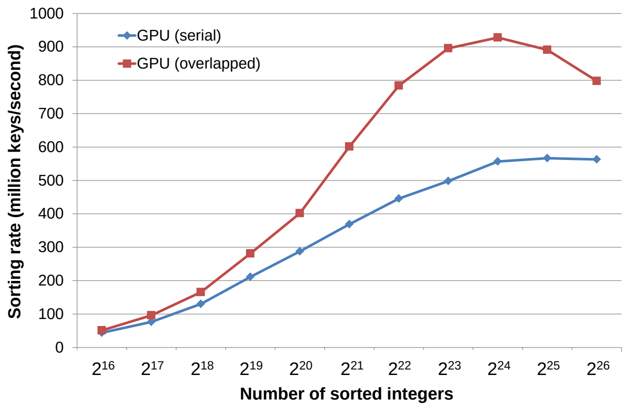
[3]`https://github.com/kylelutz/compute`
[4]`https://code.google.com/p/cudpp`

5.3 Results

The GPU radix sort implementation was tested against the highly optimised CPU radix sort found in Microsoft's Parallel Patterns Library (PPL)[5]. The `std::sort` found in Microsoft's standard template library was also added to the results as an indicator of typical sorting speeds. The keys to be sorted were generated randomly, and the results were averaged over five runs for each problem size tested.



(a) Comparison of the performance of the GPU radix sort and two CPU sorting algorithms for different problem sizes.



(b) Comparison of how overlapped transfer and execution of multiple sorts compares to simple serial scheduling.

Figure 11: Performance of the radix sort.

The performance results can be seen in Figure 11a. Despite the GPU radix sort being by far the most optimised of the GPU implementations presented here, its performance benefit over an efficient CPU solution is considerably less than the order of magnitude speedups achieved by the other solutions. For standalone sorts, the performance of our adaptation of Merrill and Grimshaw's radix sort ranged from the same speed when sorting 2^{17} elements to 3.8x faster when sorting 2^{25} elements. The results are considerably better if only the compute time of the GPU is considered. Excluding the time it takes to transfer the data to and from the GPU, the GPU's advantage over the CPU ranges from 1.8x to 9.8x over the same problem sizes. An unfortunate limitation of discreet GPUs is that the transfer of data to and from the GPU

[5]`http://msdn.microsoft.com/en-us/library/dd492418.aspx`

No. Keys	2^{16}	2^{17}	..	2^{25}	2^{26}
Transfer Overhead	35%	37%	..	59%	63%

Table 2: Data transfer contribution to the total GPU time for a number of different problem sizes.

can be a significant bottleneck, and indeed, this was the case for the GPU radix sort. For the smallest problem size tested, 2^{16}, data transfers accounted for 35% of the total GPU time, and for problem sizes greater than 2^{21}, the data transfer time actually exceeds the compute time and continually grows in proportion to compute time with larger problem sizes. This trend is shown in Table 2. The GPU's 'sweet spot' is when sorting 2^{25} elements; greater problem sizes result in a decrease in performance because the additional work no longer results in better GPU utilisation and creates more overhead from data transfers.

While the impact of data transfers on performance was severe for a standalone sort, other use cases provide opportunities to mitigate the impact of data transfers or remove the requirement completely. In use cases where the radix sort is a component of a bigger GPU program, the input data, results, or both may not need to be transferred to or from the GPU depending on where the sort is needed. For use cases where n sorts are required, the aggregate transfer time of $n - 1$ sorts can be hidden by overlapping the data transfer of queued sorts with the execution and result reading of the current sort provided the compute time exceeds the data transfer time. This is illustrated in Figure 12. Overlapped data transfer with execution

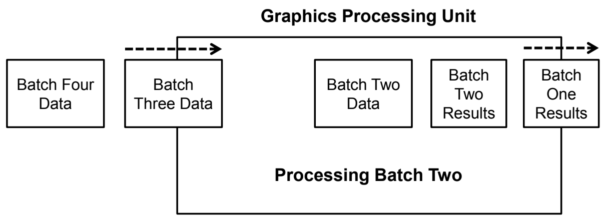


Figure 12: Illustration of how overlapped transfer and execution hides data transfer overhead.

can make a sizable performance difference if the data transfer time is high in relation to the execution time, which is the case for the radix sort. To evaluate the practical benefit of this technique, ten different sorts were scheduled on the GPU using the overlapping technique and compared to serial scheduling of the sorts. The results are shown in Figure 11b.

The results show that the performance benefit from using overlapped transfer and execution for this radix sort is very dependent on the number of items sorted. The speed improvement ranged from 1.16x when sorting 2^{16} elements to 1.8x when sorting 2^{23} elements. Larger problem sizes benefited more from this technique because the data transfer time accounted for a greater portion of the total program execution time. When sorting more than 2^{23} elements, the benefit of the technique begins to decline. This is the point where the time spent on data transfers exceeds the time spent on computation. With data transfers taking longer than execution, the execution of the next scheduled sort can no longer commence as soon as the previous sort has finished since the data needed for the sort has not yet been fully transferred to the GPU. This only worsens with larger problem sizes as they require larger data transfers. The performance benefit of the GPU radix sort is therefore largely reliant on the use case.

6 CLASSIFICATION FRAMEWORK AND ITS APPLICATION

The wealth of published success stories of gaining orders of magnitude speedups through the use of GPGPU has undoubtedly caught the attention of many scientists. However, anyone new to the field of GPGPU could be forgiven for feeling apprehensive about the steep learning curve and low-level documentation. Such characteristics are barriers to entry into the field, but this need not be so in all cases. To accelerate problems that do not map well to the GPU's architecture, it may be necessary to understand the low-level details of thread scheduling, memory transactions, and so on. For problems that map relatively well to the GPU's architecture, having such in-depth knowledge of the functioning of the GPU is not a prerequisite for obtaining a satisfactory speedup, as was found with the uncertainty model in Section 3. Consequently, it would be beneficial to be able to classify a problem as belonging to a particular GPU acceleration difficulty level, and identify guidance appropriate for that level.

6.1 Problem Difficulty Factors

The problems accelerated in Sections 3, 4, and 5 were found to require suitably different levels of knowledge of GPGPU, as explained below, to be separated into different difficulty classes.

- Accelerating the hydrological model required only a rudimentary understanding of the OpenCL API, and an efficient layout of global memory data.
- The k-difference string matching problem required additional knowledge of thread cooperation techniques, synchronisation, and GPU memory load characteristics.
- The radix sort, on the other hand, required advanced techniques and optimisations that could only be implemented by an experienced GPU developer.

Seven attributes of these problems were identified as significant factors when evaluating problem difficulty. These attributes are described briefly below.

Inherent parallelism: The ease with which the work can be distributed between hundreds to thousands of threads. Since GPUs are massively parallel devices, the less parallelism that is easily available, the harder it will be to provide the GPU with enough work and obtain a satisfactory speedup.

Branch divergence: Within compute units, wavefronts are executed by SIMD stream processors. Consequently, *branch divergence* (or control flow divergence) within a wavefront results in lower GPU utilisation, as stream processors that do not follow a branch are forced to idle [17]. Solutions that result in poor performance due to branch divergence may need to be refactored to eliminate the divergence or to incorporate techniques such as branch fusion [17].

Problem size: This is the amount of work that can be parallelised. If managed correctly, an abundance of work enables the GPU to hide much of the memory access latency (given the resource constraints) [18]. Conversely, a low amount of parallel work increases the likelihood of compute units stalling on pending memory operations, or idling without work. Increasing instruction-level parallelism (ILP) can help to improve low GPU utilisation due to insufficient alternative work [19].

Required computational parallelism: We define *required computational parallelism (RCP)* as the number of wavefronts or warps required to cover memory access latency. This is calculated as $RCP = (a + m) \ / \ a$, where a is the total arithmetic latency and m is the total global memory latency for a single work-item. This is identical to *computation warp parallelism* [20]; we have used a different name to avoid association with a particular brand of GPUs (*warp* is a CUDA term). RCP is similar to arithmetic intensity, except that it relates to time rather than quantity. Programs with a low RCP are preferable as this simplifies the hiding of memory latency. High RCP values mean that typically more effort is required to add thread-level parallelism (TLP) and ILP to hide memory latency and ensure efficient memory requests.

Memory access pattern regularity: GPUs are able to provide the highest memory bandwidth when memory access patterns are regular and have high spatial locality. Irregular access patterns or patterns with low spatial locality prevent memory requests from being coalesced into fewer memory transactions, or result in unbalanced utilisation of the memory controllers [18, 21]. This can sometimes be addressed by preprocessing the input data to group similar data inputs, or rearrangement of the data items within a work-group.

Data transfer overhead: In the context of running a standalone GPU program, the time taken to transfer data to and from the GPU is pure overhead. Thus, GPU programs with large data requirements will be harder to accelerate than similar GPU programs with smaller data requirements, and may require effort to reduce the amount of data transferred or improve the data transfer speed.

Thread cooperation: The difficulty of thread cooperation lies in its use of local and global memory, and synchronisation primitives such as barriers and atomic operations. Local and global memories are orders of magnitude slower than private memory, and atomic operations serialise concurrent memory requests. Thus, careful thought must be given to the granularity and structure of thread cooperation to ensure that performance penalties are minimised.

6.2 Classification Framework

For the purposes of the initial classification framework, ordinal ratings are used for all the difficulty indicators, namely, ‘Negligible’, ‘Low’, ‘Moderate’, or ‘High’. These ratings have different meanings for the different difficulty indicators, as discussed below.

Inherent parallelism: ‘Negligible’ means there is no inherent parallelism. A ‘Low’ rating means that inherent parallelism is not linked to problem size, and there is an insufficient number of parallel tasks for the number of processors on the GPU. A ‘High’ rating means there is an abundance of parallel tasks, and the inherent parallelism is typically linked to problem size.

Branch divergence: ‘Negligible’ means there is virtually no branch divergence, while ‘High’ denotes an abundance of branch divergence.

Problem size: ‘Negligible’ means there are virtually no work tasks. A ‘Low’ rating means the envisaged problem size is not sufficiently large to provide work for all the stream processors in the GPU, or not large enough to take advantage of TLP or ILP. A ‘High’ rating means the problem size allows for an abundance of TLP and ILP.

Required computational parallelism: A ‘Negligible’ RCP means no TLP is needed, while a ‘High’ RCP means it is difficult or impossible to hide memory access latency.

Memory access regularity: ‘Negligible’ means memory transactions are completely irregular and have no spatial locality, whereas ‘High’ means memory accesses are predominantly or always regular and have high spatial locality.

Data transfer overhead: ‘Negligible’ means the data transfer overhead is not a consideration, whereas ‘High’ means the data transfer overhead contributes significantly to overall program execution time.

Thread cooperation: ‘Negligible’ means there is virtually no thread cooperation, while ‘High’ means thread cooperation is prevalent throughout the solution.

These difficulty indicators were used to construct a difficulty classification framework to provide the user with an idea of overall problem difficulty, particularly for extreme cases where problem acceleration is either very simple or very difficult. To quantify the overall difficulty rating, the four ordinal ratings are coded, respectively, as 0,1,2,3 or 3,2,1,0 for each difficulty factor depending on whether a low or high rating is preferable for parallelising the problem.

Table 3: Classification of the hydrological uncertainty model.

Difficulty Indicator	Rating			
	Negligible	Low	Moderate	High
Inherent Parallelism				✔ 0
Branch Divergence	✔ 0			
Problem Size				✔ 0
Required Computational Parallelism			✔ 2	
Memory Access Pattern Regularity			✔ 1	
Data Transfer Overhead	✔ 0			
Thread Cooperation	✔ 0			

6.3 Classification of Accelerated Problems

Given that the classification framework was modelled on what was learned from accelerating the three case studies, evaluation of these problems using the framework should provide a difficulty estimation similar to what was actually experienced.

6.3.1 Case Study 1: Hydrological uncertainty model

Inherent parallelism: Many independent instances of the model are run with different input data, with each instance regarded as a parallel task to be run as a separate work-item. Thus, parallelism is related to problem size and the problem is considered to be embarrassingly parallel.

Branch divergence: There is unlikely to be much branch divergence between ensembles since the primary loop iteration counts are identical, and the data-dependent branches typically have a low depth. Kernel profiling confirmed this by indicating an average branch divergence of under 3%.

Problem size: Since the purpose of this program is uncertainty analysis, the parameters for each model run are generated randomly (within certain bounds). The problem size is therefore only restricted by the number of possible parameter configurations; this number is large enough not to be of concern.

RCP: This ratio was evaluated to be 4.7. We consider this to be ‘Moderate’, as it would require TLP amounting to just under half the maximum number of wavefronts that can be scheduled on a SIMD unit.

Memory access pattern regularity: Other than a small amount of branch divergence, the memory access pattern is regular. However, the memory access locality is low because of the large data structures that store the model data. Thus, we classify this as ‘Moderate’.

Data transfer overhead: The data transfer to the GPU consists of the core model data and the parameter sets for each ensemble. Since the parameter sets are small, the quantity of input data transferred between the host and the GPU does not scale significantly with problem size. With model runs being computationally intensive, the data transfer overhead should only constitute a small portion of the overall GPU execution time. If the GPU speedup factor is estimated at 10x or 100x, the data transfer overhead is estimated at less than 0.1% for a problem size of 50,000 ensembles. We therefore classify this as ‘Negligible’.

Thread cooperation: Each work-item calculates its own model ensemble, and thus no thread cooperation is required.

Table 3 shows the difficulty classification of the hydrological uncertainty model. The only non-zero difficulty indicators are RCP (with a coded value of 2) and memory access pattern regularity (coded as 1). The overall difficulty according to this evaluation is 3, which suggests a relatively low acceleration difficulty.

6.3.2 Case Study 2: K-Difference String Matching

Inherent parallelism: Each k-difference comparison can be run independently of all others, and the number of comparisons is linked to problem size resulting in ‘High’ inherent parallelism.

Branch divergence: Comparisons between strings of different lengths within a wavefront will result in some work-items finishing before others. Coupled with Ukkonen’s cut-off, this results in high branch divergence. According to kernel profiling, the average amount of branch divergence is over 55%.

Problem size: The number of strings compared is expected to be in the millions for practical applications of this program. There is thus an abundance of parallel work available.

RCP: The RCP for comparing short and long strings was calculated as 3.4 and 2.7, respectively, which we consider to be ‘Low’.

Memory access pattern regularity: The original solution used a linear layout for storing the string data and partial results in memory. This resulted in low memory access locality as well as low memory access regularity, thus making this classification ‘Low’.

Data transfer overhead: The input consists of the strings to be compared, and the output for each comparison is an integer corresponding to a positive or negative match. If a 10x speedup is assumed with a problem size of 2,000 short test patterns and 8,192 short input strings, the transfer overhead is estimated at 2.22%, which is classified as 'Low'.

Thread cooperation: Inter-group cooperation between work-items is not needed in this algorithm, but can be recognised as greatly beneficial for data sharing purposes. Since this may not be known prior to implementation, this has been classified as 'Negligible'.

Unlike the hydrological uncertainty ensemble model, there are many non-zero difficulty indicators in this classification. When summed, these give an overall difficulty of 7.

6.3.3 Case Study 3: Radix Sort

Inherent parallelism: Parallelism is obtained by partitioning the key space between a number of work-items and sharing information at key points in the algorithm. There are many sections of code in which thread cooperation occurs involving a limited number of work-items, which reduces the inherent parallelism to 'Moderate'.

Branch divergence: As mentioned above, there are many sections of code in which a limited number of threads participate, which implies a moderate amount of branch divergence. This is supported by kernel profiling.

Problem size: GPU radix sorting is typically only required for very large numbers of keys, which means problem size is 'High'.

RCP: The average RCP for the primary kernels[6] was calculated as 11. This is 'High', since it exceeds the maximum number of wavefronts that can be scheduled on a SIMD unit.

Memory access pattern regularity: Irregular access patterns are found in the final kernel when writing the results of a sorting pass to global memory. Given that this code is visited a number of times from multiple sorting passes, this algorithm has been classified as having 'Moderate' memory access pattern regularity.

Data transfer overhead: Other than several non-vector input variables, the input and output data sizes are identical. For a problem size of 2^{26} 32-bit integers, the data transfer overhead is estimated at 59%, which is easily classified as 'High'.

Thread cooperation: A significant amount of both intra- and inter-group thread cooperation is used in this algorithm to enable parallel execution.

With an overall difficulty of 13, it is clear that the radix sort is a much harder problem to accelerate than the first two problems.

[6]The second kernel was omitted from the calculation because it is designed to only use a single work-group.

6.3.4 Reflection

Classification of the three case studies using the proposed framework resulted in difficulties of 3, 7, and 13, respectively. Relative to each other, these ratings correspond well to the actual implementation difficulties experienced. Furthermore, the individual difficulty indicator ratings correspond to either the evaluation of the knowledge required for replicating our GPU solutions, or identified bottlenecks. This shows that the evaluation of the identified difficulty indicators gives some indication of the difficulty of problem acceleration using GPUs.

7 CONCLUSION

A current problem with GPGPU is that it can be seen as a field that requires a great deal of technical knowledge to be successful. Certain classes of problems do indeed require this, but others can be accelerated with very little technical knowledge of GPUs. However, it may be difficult for inexperienced GPU programmers to make this distinction. This study aimed to identify the attributes of problems that are important in determining the level of difficulty of accelerating problems on a GPU in order to create a classification framework that can be used to provide an initial estimate of the difficulty involved in doing so. This was done by accelerating three problems, perceived to have different difficulty levels, on a GPU. The relatively high correlation of the actual effort involved in accelerating each problem and the overall difficulty value predicted by applying the classification framework to the respective problem shows that our classification approach has potential.

However, there are a number of limitations that need to be addressed in future revisions, including obtaining reliable quantitative evaluation methods for the difficulty indicators and investigating whether the difficulty impact of some indicators is more significant than that of others.

8 ACKNOWLEDGEMENTS

This work was undertaken in the Distributed Multimedia CoE at Rhodes University, with financial support from Telkom SA, Tellabs, Genband, Easttel, Bright Ideas 39, THRIP and NRF SA (UID 75107). The authors acknowledge that opinions, findings and conclusions or recommendations expressed here are those of the author(s) and that none of the above mentioned sponsors accepts liability whatsoever in this regard.

REFERENCES

[1] M. Daga, T. Scogland and W. Feng. "Architecture-Aware Mapping and Optimization on a 1600-Core GPU". In *Proceedings of the 2011 IEEE 17th International Conference on Parallel and Distributed Systems*, ICPADS '11, pp. 316–323. IEEE Computer Society, 2011.

[2] M. Garland and D. B. Kirk. “Understanding throughput-oriented architectures”. *Commun. ACM*, vol. 53, pp. 58–66, Nov. 2010.

[3] K. Fatahalian and M. Houston. “GPUs: A Closer Look”. *Queue*, vol. 6, no. 2, pp. 18–28, Mar. 2008.

[4] J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone and J. C. Phillips. “GPU Computing”. *Proceedings of the IEEE*, vol. 96, no. 5, pp. 879–899, May 2008.

[5] Advanced Micro Devices. *AMD Accelerated Parallel Processing OpenCL Programming Guide*, 2012.

[6] Khronos OpenCL Working Group. *The OpenCL Specification, version 1.2, Rev 15.* Khronos Group, 15 November 2011.

[7] H. Moradkhani and S. Sorooshian. “General Review of Rainfall-Runoff Modeling: Model Calibration, Data Assimilation, and Uncertainty Analysis”. In *Hydrological Modelling and the Water Cycle*, vol. 63 of *Water Science and Technology Library*, pp. 1–24. Springer Berlin Heidelberg, 2008.

[8] D. A. Hughes, E. Kapangaziwiri and T. Sawunyama. “Hydrological model uncertainty assessment in Southern Africa”. *Journal of Hydrology*, vol. 387, no. 3–4, pp. 221–232, 2010.

[9] T. Yen and M. K. Reiter. “Traffic Aggregation for Malware Detection”. In *Proceedings of the 5th international conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, DIMVA ’08, pp. 207–227. Springer-Verlag, Berlin, Heidelberg, 2008.

[10] F. Aleen and K. Mahalingam. “Improving Bayesian Spam Filters Using String Edit Distance Algorithm”. In *International Conference on Internet Computing*, pp. 121–125. CSREA Press, 2008.

[11] G. Navarro. “A guided tour to approximate string matching”. *ACM Comput. Surv.*, vol. 33, no. 1, pp. 31–88, Mar. 2001.

[12] E. Ukkonen. “Finding approximate patterns in strings”. *Journal of algorithms*, vol. 6, no. 1, pp. 132–137, 1985.

[13] G. Myers. “A fast bit-vector algorithm for approximate string matching based on dynamic programming”. *J. ACM*, vol. 46, no. 3, pp. 395–415, May 1999.

[14] H. Hyyrö. “Explaining and Extending the Bit-parallel Approximate String Matching Algorithm of Myers”. Tech. rep., Dept. of Computer and Information Sciences, University of Tampere, 2001.

[15] D. Merrill and A. Grimshaw. “High Performance and Scalable Radix Sorting: A case study of implementing dynamic parallelism for GPU computing”. *Parallel Processing Letters*, vol. 21, no. 2, pp. 245–272, 2011.

[16] N. Satish, M. Harris and M. Garland. “Designing efficient sorting algorithms for manycore GPUs”. In *Proceedings of the 2009 IEEE International Symposium on Parallel & Distributed Processing*, IPDPS ’09, pp. 1–10. IEEE Computer Society, Washington, DC, USA, 2009.

[17] B. Coutinho, D. Sampaio, F. M. Q. Pereira and W. Meira Jr. “Divergence Analysis and Optimizations”. In *Proceedings of the 2011 International Conference on Parallel Architectures and Compilation Techniques*, PACT ’11, pp. 320–329. IEEE Computer Society, Washington, DC, USA, 2011.

[18] Advanced Micro Devices. “AMD Accelerated Parallel Processing OpenCL Programming Guide”. Online: `http://developer.amd.com/wordpress/media/2013/08/AMD_Accelerated_Parallel_Processing_OpenCL_Programming_Guide.pdf` [Accessed 01/08/13], August 2013.

[19] V. Volkov. “Better performance at lower occupancy”. In *Proceedings of the GPU Technology Conference, GTC*, vol. 10. 2010.

[20] S. Hong and H. Kim. “An analytical model for a GPU architecture with memory-level and thread-level parallelism awareness”. *SIGARCH Comput. Archit. News*, vol. 37, no. 3, pp. 152–163, Jun. 2009.

[21] J. Meng, V. A. Morozov, K. Kumaran, V. Vishwanath and T. D. Uram. “GROPHECY: GPU performance projection from CPU code skeletons”. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, p. 14. ACM, 2011.