A Case Against Small Data Types in GPGPUs

Ahmad Lashgar
Department of Electrical and Computer Engineering
University of Victoria
Email: lashgar@uvic.ca

Amirali Baniasadi
Department of Electrical and Computer Engineering
University of Victoria
Email: amiralib@uvic.ca

***Abstract*—In this paper, we study application behavior in GPGPUs. We investigate how data type impacts performance in different applications. As we show, expectedly, some applications can take significant advantage of small data types. Such applications benefit from small data types as a result of increasing cache effective capacity, reducing memory pressure, access latency, and memory bandwidth demand. This typical behavior, however, has some exceptions.**

In this work we show that although using small data types can improve memory efficiency, it can also degrade performance due to an increase in the number of cache miss handling stalls. We present *1D stencil* application as a case example where this occurs.

We analyze our findings through a combination of real-hardware and cycle-accurate simulation. Studying regular highly-coalesced memory pattern, we conclude that cache miss handling resources can play an important role in negating small data type advantages.

I. Introduction

Typical data-parallel applications perform the same operation on many independent elements iteratively. Such individual elements can be particles' masses in an N-body simulation [1], elements of a large matrix in *matrix-matrix multiplication* [3] or pixels in 3D rendering [2].

Several parameters can impact the developers' decision regarding the data type used for data elements. One key factor is calculation precision. This sets a lower-bound on the number of essential bytes to avoid calculation overflow/underflow. Other impacting issues are cache and memory subsystem considerations. Smaller data types may increase the effective capacity of on-chip caches, as space is used more conservatively. Additionally, smaller data types can lower memory bandwidth demand and reduce memory latency. Another impacting parameter is instruction delay. Although 4-, 2-, and 1-byte integer operations have the same delay in most general-purpose processors, the delay of 4-byte and 8-byte floating point operations differ significantly [13]. This can further motivate developers to use imprecise yet faster data types and refine the precision through extra operations [5] in compute-bounded applications. In general, deciding the elements' data type comes with a tradeoff between performance (memory efficiency and instruction latency) and calculation precision.

In this study, we relax the precision constraint and investigate how data type decisions can affect performance in GPGPU. We study three different benchmarks from different application domains and as representatives of typical data-parallel applications. We include applications from image processing, linear algebra, and computer graphics. Intuitively we would expect that small data type sizes outperform large data type sizes consistently due to less bandwidth and cache pressure. However, as we show in this study, there are cases where using small data types degrades performance. We analyze this behavior and provide insight regarding the reasons behind it. Exploring different behaviors, we investigate the miss handling organization used in GPGPUs. We investigate critical miss handling resources and discuss scenarios that can stall load/store units in these processors.

In summary, we make following contributions:

- In GPGPUs, we study three applications, *matrix-matrix multiplication*, *1D stencil*, and *stereo matching*, under different data type sizes. We show smaller data types may improve or degrade performance.
- We show that data type can impact the number of core stalls significantly, leading to a huge impact on performance. We show data type variations make such impact by affecting MSHR (Miss Status Holding Register) table contention or memory coalescing capabilities.
- We study *1D stencil* test case, a highly-regular memory-bounded application as an example of when using smaller data types fails. We analyze the behavior of this test case under various data type sizes and measure memory efficiency, instruction throughput, and core stalls.
- We use a combination of real hardware measurements and simulations to analyze the test case. First, we show how our simulation follows the performance trend of real hardware, when executing the test case under different data types. Then, we use simulations to investigate hardware bottlenecks under different data type sizes. We also exploit simulation flexibility to evaluate performance under various microarchitecture changes; including different MSHR/cache configurations.

The remainder of this paper is organized as follows. In Section II, we report how applications react to different data sizes. In Section III, we present an overview of the baseline GPGPU. In Section IV, we discuss different types of stalls that may occur upon handling a memory request. In Section V, we introduce our implementation of *1D stencil*. In Section VI, we present our methodology for evaluations. In Section VII, we present experimental results analyzing *1D stencil* under different data type sizes. In Section VIII, we discuss how our findings vary under different architectures or memory patterns. In Section IX, we review related works. Finally, in Section X, we discuss concluding remarks.

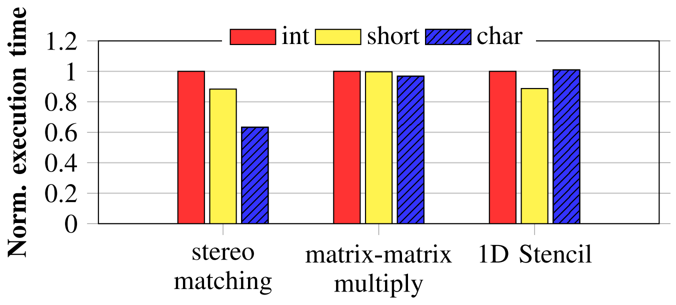

Figure 1. Execution time of different applications under varying data type sizes.

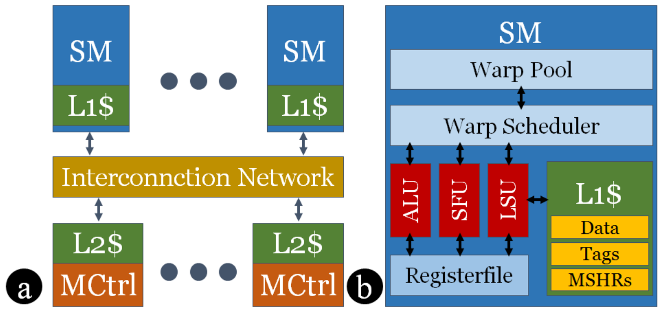

Figure 2. (a) High-level design of a GPU chip. (b) microarchitecture of SM.

II. Applications and Data Types

In this section, we discuss three different applications and how they behave under varying data type sizes. We evaluate *matrix-matrix multiplication*, *1D stencil*, and *stereo matching*. We have implemented each benchmark in CUDA.

Matrix-matrix multiplication: In *matrix-matrix multiplication*, every working thread calculates one element of the output matrix. Hence, every thread reads i) one entire row of the first input matrix and ii) one entire column of the second input matrix.

1D stencil: In every iteration of *1D stencil*, every working thread computes one element in the output. Each thread stores the sum of three neighbor elements in the input at the output array.

Stereo matching: We perform *stereo matching* through belief propagation algorithm [4]. In each iteration, 16 labels are propagated to right, left, up, and down directions, respectively. Upon horizontal propagation, each thread performs the operations for one entire row while upon vertical propagation, each thread performs the operations for one entire column.

The basic element of operations in *matrix-matrix multiplication*, *1D stencil*, and *stereo matching* is an element in matrix, an element in array, and a label in disparity map. We evaluate the performance of these benchmarks under varying basic element's data type sizes. We assume three basic element sizes: 4-byte integer (int), 2-byte integer (short), and 1-byte integer (char). In order to do this, we re-compile each benchmark three times, once for each data type.

Figure 1 reports the kernel's execution time for *matrix-matrix multiplication*, *1D stencil*, and *stereo matching* under varying data type sizes (refer to Section VI for the methodology). Under *stereo matching* workload, smaller data types reduce execution time significantly. In this workload, using char data type reduces the execution time by 37%. Although the same trend can be seen in *matrix-matrix multiplication*, the performance impact is much lower (less than 3% gap between int and char). Under *1D stencil*, however, we see a different trend; optimal data type for performance is short which is a type between the largest and smallest data type. short data type outperfoms int and char by 11% and 12%, respectively.

In the remainder of this paper, we investigate the underlying GPGPU-like microarchitecture and memory pattern of *1D stencil* to find the reason behind this observation.

III. Background

CUDA workload is composed of millions of threads. To reduce inter-thread communications, threads are restricted to communicate with threads within the same thread-block. Thread-blocks are the coarse scheduling elements whose threads execute on the same Streaming Multiprocessors (SMs) concurrently. A GPU may have one or more SMs and CUDA workload should have enough thread-blocks to occupy the entire GPU.

Figure 2a shows the microarchitecture of a typical GPU. SMs send memory requests to the on-chip NoC and fetch memory lines from memory controllers. Memory bandwidth can be very high (150 to 320 GB/s) and memory access latency takes hundreds of cycles (100 to 1000 cycles). Figure 2b shows the microarchitecture of an SM. Each SM is a SIMD deep-multithreaded pipelined processor.

SM interleaves the execution of thousands of threads. In order to reduce the scheduling overhead in the pipeline front-end, threads are grouped into coarser scheduling elements referred to as warps. Threads within a warp are processed in lock-step, enhancing efficiency for the SIMD units employed in the pipeline backend. Each cycle, the warp scheduler issues a ready instruction from a warp to ALU, SFU, or LSU. ALU (executing logical/arithmetic operations) and SFU (executing special functions, e.g. logarithm) are SIMD units (256 to 2048-bit wide). Upon execution, every individual thread of a warp occupies one SIMD lane of ALU/SFU. LSU executes load and store instructions. LSU can access one cache line every core cycle and can coalesce the memory accesses of the threads within the same warp (if accessing the same cache line). Otherwise, LSU serves threads serially.

LSU can access memory hierarchy to fetch the requested data. First, LSU sends a request to the highest level, per SM L1 cache. Upon an L1 miss, data should be fetched from lower memory levels. Under such circumstances, LSU reserves one cache line to fill in the fetched data. It also reserves one MSHR entry per outstanding memory access (every L1 cache has a few dedicated MSHRs). Then, it sends a request for the required data to the lower memory level, L2 cache in this microarchitecture. LSU pipeline may stall upon an L1 cache miss due to the lack of enough cache lines or MSHR resources (number of MSHRs or possible mergers). In the next section, we discuss how such stalls may occur. We also discuss how these stalls relate to the memory access pattern and size of data type.

IV. Cache Miss Handling Stalls

Upon executing a warp of memory threads, LSU groups thread accesses to the same cache line and satisfies each request group one-by-one serially. After grouping, LSU performs the following operations to satisfy each memory request group.

The process starts by looking up the requested tag line in the L1 cache. If the line exists in the cache, it can be fetched and the request is satisfied. Otherwise, before sending a memory request, LSU lookups MSHRs for an outstanding memory request to the same line. If the same request exists and the MSHR entry has a free field to merge another memory request, LSU merges the pending request with the existing request. If there isn't a free merger field, LSU stalls the pipeline and waits for a free field. If there is a free merger field, no further operation is required and the request is satisfied. Otherwise, if there is no existing request for the same line, LSU should reserve i) one cache line and ii) one MSHR entry for the new request. If either of reservations fail, LSU pipeline stalls and waits for the resource to be freed.

The number of outstanding accesses supported by the cache determines maximum memory-level parallelism (MLP [14]) that it can provide. For every outstanding cache miss, LSU reserves an MSHR entry to track the request. Moreover, it reserves a cache line that is to be filled upon the arrival of data from lower memory levels. Under best case scenario, the concurrent number of outstanding memory requests per SM is:

$$min(\#cacheLines, \#MSHRs)$$

where #cacheLines and #MSHRs refer to the number of lines in the L1 cache and the number of entries in the MSHR table, respectively. However, the number of outstanding memory requests often exceeds this minimum as each MSHR entry can be shared (or merged) with a few other memory requests, accessing the same line. Therefore, maximum outstanding memory requests per SM is:

$$\#MSHRMergers \times min(\#cacheLines, \#MSHRs)$$

where #MSHRMergers refers to the number of merger fields in each MSHR entry. A more challenging scenario happens once all outstanding cache misses are mapped to the same set. Accordingly, the maximum outstanding memory requests is limited by the number of ways in a cache set, as indicated by the following:

$$\#MSHRMergers \times min(\#wayInaSet, \#MSHRs)$$

where #wayInaSet refers to the number of ways available in each L1 cache set. And finally, the worst case scenario happens once all outstanding cache misses are mapped to the same set and need the same line too. Note that in this case the cache can only have #MSHRMerger outstanding memory requests. At this worst case, due to structural hazards, warps are stalled in LSU. They also may stall dependent instructions due to the data dependency, dropping ALU and SFU utilization and degrading performance. Figure 3 illustrates an example on how the worst case scenario may occur.

The worst case scenario may occur more frequently upon performing operations on data types which are small in size, e.g. 1-byte per element. This happens when warps of the same thread-block are operating on the neighbor small data elements. Although performing on adjacent elements improves data locality and cache/memory efficiency, warps of the same thread-block generate a higher number of memory accesses mapping to the same cache line. These warps send a request for the same cache line and occupy the same MSHR. Due to the limited capacity of each MSHR entry, it cannot merge all accesses. Therefore, some of the warps stall at LSU and wait for resource availability (MSHR merger field in this case). These stalls can be significant and even harm the memory bandwidth/latency gain that is coming from small data types. In the next section, we investigate this further and show how it harms performance in *1D stencil* test case.

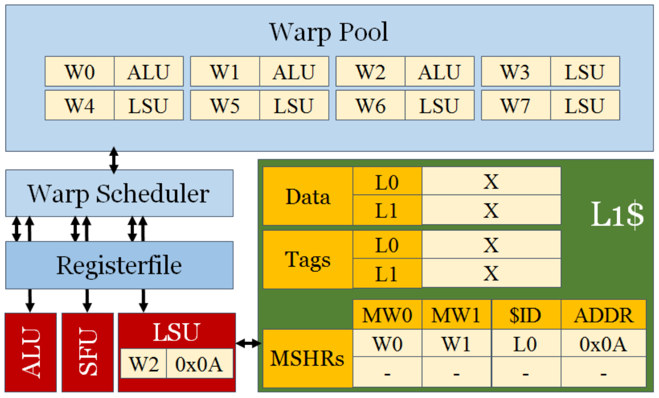

Figure 3. Example of contention on MSHRs that leads to significant number of stalls. microarchitecture details are excluded for simplicity. In this example, there are eight concurrent warps, two L1 cache lines and two MSHR entries. Each MSHR entry records the requested address (ADDR) and the reserved cache line ($ID). Also each entry can merge requests of up to two warps (MW0 and MW1), if the two need the same line. In this snapshot, one MSHR entry is reserved for W0 and W1. The entry tracks the request for 0x0A from lower memory levels. Assume the next instruction of these warps are i) ALU operation and ii) stalled due to data dependency (as depicted in Warp Pool by an ALU word nearby the warp). LSU calculates the address for next warp, W2. W2 needs 0x0A, the same line as W0 and W1. LSU cannot merge W2's request with the existing one since both merger fields are already occupied (by W0 and W1). Meanwhile, although there is an empty entry in MSHRs, LSU cannot book W2's anywhere else in MSHRs table. This is because one memory line is not allowed to be placed in two different ways of a cache. Therefore, LSU stalls its pipeline and waits until it satisfies W2's request. Looking at Warp Pool, we find W3 to W7 which are ready to execute an LSU operation. However, stall of LSU pipeline leads to a structural hazard and delays W3, W4, W5, W6, and W7. As illustrated in this example, occurrence of the worst case scenario can stall many warps.

V. Case Study: 1D Stencil

We investigate *1D stencil* operation as a case where using small data types harms performance. To do so, we study the impact of data type size on performance, memory bandwidth/latency, and MSHR merging stalls.

Equation 1 defines *1D stencil* operations calculating the average of three elements as the output for each element.

$$a[i] = \begin{cases} (a[i-1] + a[i] + a[i+1])/3 & \text{if } 1 < i < n \\ a[i] & \text{otherwise} \end{cases} \quad (1)$$

We assume n elements for the input/output array. In our implementation, two distinct arrays are allocated for input and output. *1D stencil* is an iterative algorithm, repeating Equation 1 for certain number of iterations. Since these operations

Table I. GTX 480 CARD SPECIFICATIONS.

Compute Capability	2.0
DRAM size	1535 MB
SMs	15
CUDA cores per SM	32
Core clock	1.40 GHz
Memory clock	1.8 GHz
Memory bus width	384-bit
L2 Cache Size	786 KB
Warp size	32
Max. threads per block	1024

Table II. GPGPU-SIM CONFIGURATIONS FOR MODELING GTX 480.

GPU chip	
SMs	15
Memory controllers	6
Sub partition / memory controller	2
L1 Cache / SM	
Size	16KB
# sets	32
# ways / set	4
line size	128 Bytes
# MSHR entries	32
# MSHR merger fields / MSHR entry	8
L2 Cache / sub partition	
Size	64KB
# sets	64
# ways / set	8
line size	128 Bytes
# MSHR entries	32
# MSHR merger fields / MSHR entry	4

are performed iteratively, all array elements will eventually communicate with each other indirectly. The memory access pattern associated with this application is highly regular and does not change over different iterations.

We evaluate *1D stencil* under different input/output data types; 4-byte, 2-byte, and 1-byte array elements. Using smaller data types for array elements can save memory bandwidth. This is due to the fact that the same operations are performed while fetching fewer data from memory (compared to larger data types). Additionally, smaller data types can save memory latency since fewer memory requests are made. At the downside, small data types may exacerbate MSHR merging stalls since many warps need the same cache line. For example, consider a benchmark where adjacent threads access consecutive array elements. Under such circumstances, assuming 1-byte array elements, 128-byte cache lines, and 32-thread warps, 128 consequent threads (four warps) need their data from the same cache line. As we illustrated in Figure 3, these warps, if executed back-to-back, may occupy the MSHR merger fields and stall LSU and other concurrent warps. In the next section, we evaluate this issue in depth and report the memory bandwidth/latency gains and MSHR merging stalls.

VI. METHODOLOGY

In Section VII, we evaluate performance of *1D stencil*. Our evaluations include measuring execution time, instruction throughput, total number of memory accesses, DRAM bandwidth utilization, memory access latency, L1 cache miss rate, and core stalls. We exploit both real and simulated GPUs to perform our evaluations. While the real GPU is used to measure performance, we perform cycle-accurate simulations to analyze parameters impacting performance further in-depth. Also simulation's flexibility allows studying and investigating

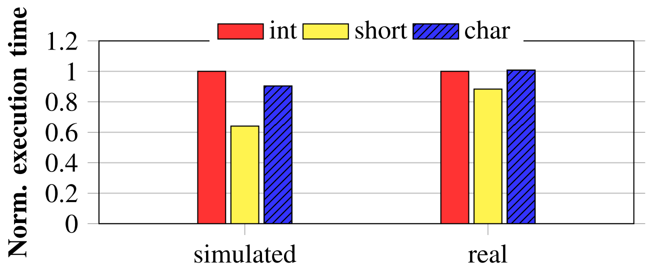

Figure 4. Normalized execution time of 1D stencil under real/simulated GTX 480 and different data types (int, short, and char are 4-, 2-, and 1-byte per element).

the impact of increasing/decreasing the resources. We use NVIDIA GeForce GTX 480 card as the real GPU in our evaluations. Table I reports the card specifications. The system has 12 GB of main memory RAM and one Intel Core i7 960 as CPU. We use GPGPU-sim 3.2.2 [6], cycle-accurate performance and power simulator, to model GTX 480. We use the configuration files provided publicly for GTX 480 in the GPGPU-sim package. Table II reports configuration parameters which are used in our evaluations.

We have implemented *1D stencil* algorithm in CUDA [7]. We assume 256 threads (or eight warps) per thread-block. Also we run the algorithm for four iterations and 512K elements in array. We use CUDA 4.0 as the runtime environment. For performance evaluations, we report the kernel execution time and exclude GPU initializations and memory transfers. On real hardware evaluations, we use CUDA Profiler [8] to measure the kernel time. For the simulated GPU, we report the number of simulation cycles taken to execute the kernel. For real hardware, every reported number is the harmonic mean of 30 different runs.

VII. EXPERIMENTAL RESULTS

In this section, we investigate *1D stencil* under different data types.

A. Performance

Figure 4 reports the normalized execution time of *1D stencil* with different data types under real and simulated GTX 480. int bars report *1D stencil* time under 4-byte elements for input/output array. Similarly, short and char bars report the time under 2-byte and 1-byte elements, respectively. Unlike *matrix-matrix multiplication* and *stereo matching* where performance improves upon moving to shorter data types, the optimal data type in *1D stencil* is in the middle. The same trend can be seen under both real hardware and simulator evaluations; performance starts to improve upon moving from 4-byte (int) to 2-byte (short) elements and degrades upon moving from 2-byte (short) to 1-byte (char) elements.

B. In-depth analysis

In this section, we further analyze *1D stencil* under GTX 480 simulation. Figure 5 reports the memory efficiency metrics, including total number of memory requests, DRAM bandwidth utilization, average memory access latency, and L1 cache miss rate, under different data types. As reported,

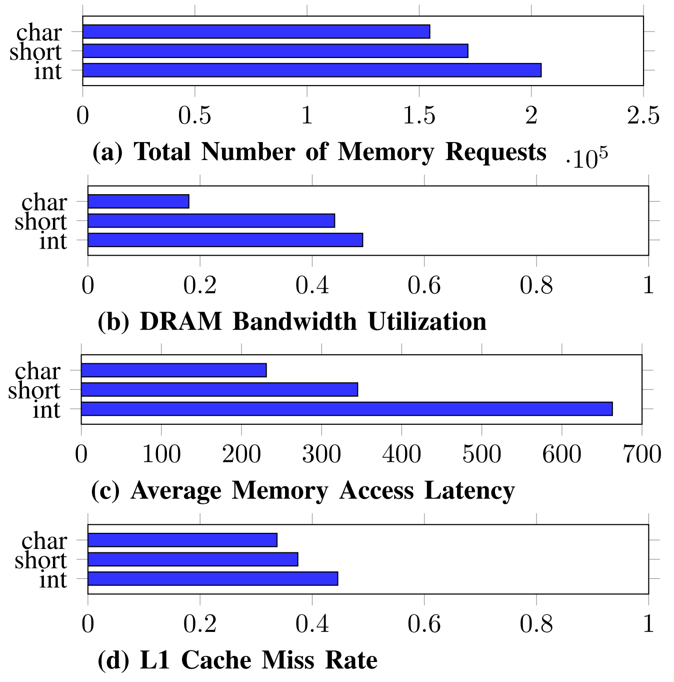

Figure 5. Memory efficiency metrics under different data types for 1D stencil.

smaller data type sizes achieve higher memory efficiency. Total number of memory accesses made by smaller data types is lower than larger data types. For example, using char data type reduces memory accesses by 24%, compared to int. Accessing memory infrequently can drop DRAM bandwidth utilization and average memory latency. As reported in Figure 5 (b) and (c), using char data type reduces DRAM bandwidth utilization and average memory latency by 31% and 65%, respectively, compared to int. Using smaller data types, the demanded data of thread-blocks fits in fewer cache lines. Therefore, smaller data types can improve the effective cache capacity. As reported in Figure 5 (d), smaller data types reduce L1 cache miss rate by 8%-11%.

Despite the significant gain in memory efficiency, smaller data types may increase congestion on single cache line and MSHR entry. To investigate this congestion, we report the breakdown of total number of stalls in Figure 6. Legends indicate the number of cycles that the LSU is stalled due to:

- **nomerger:** lack of enough merger fields in MSHR entry.
- **nomshrline:** lack of free MSHR lines.
- **nocacheline:** lack of enough L1 cache lines to reserve for outstanding cache misses.
- **coalescing:** access serialization to different cache lines, within the same warp.

Moving from 4-byte (int) to 2-byte (short) and 1-byte (char), we observe a significant increase in the number of nomerger stalls and significant decrease in coalescing stalls. Smaller data types place more array elements into one cache line. This increases the rate of concurrent requests for the same cache line from different threads. Hence, threads' memory accesses congest on an MSHR entry and occupy all of its MSHR merger fields. This explains why we observe a significant number of nomerger stalls for smaller data types. Meanwhile, having more array elements in the same cache line reduces the number of coalescing stalls. Under smaller data types in *1D stencil*, threads of a warp are more likely to need the same cache line, reducing the number of cache access serialization and coalescing stalls.

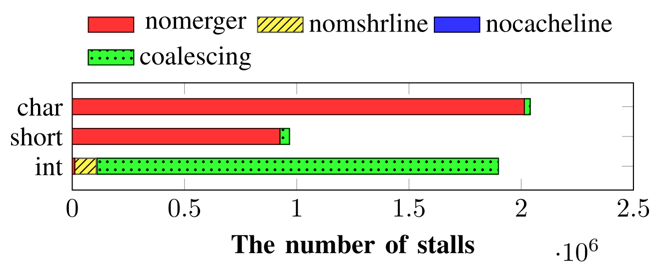

Figure 6. Breakdown of stalls under *1D stencil* for different data types.

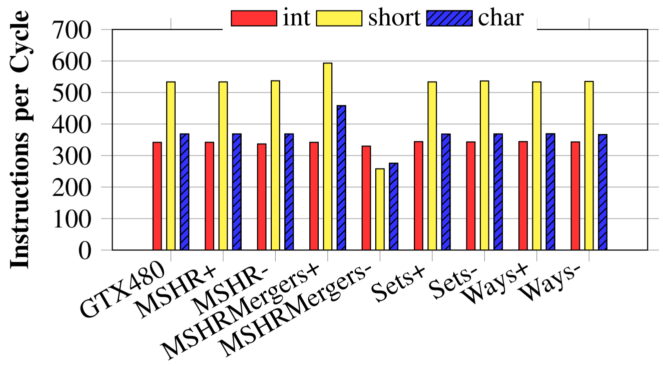

Figure 7. Performance potential behind various cache and memory fetching resources. *MSHR*, *MSHRMergers*, *Sets*, and *Ways* indicate a change in number of MSHR entries, MSHR merger entries, L1 sets, and L1 ways in set, respectively. + or - indicate 2X higher or lower resources, respectively.

C. *Performance potential*

Figure 7 reports performance when increasing/decreasing various cache or memory fetching resources. The first group on left reports performance under baseline GTX 480. Each remaining group has almost the same configuration as GTX 480, except for doubling (denoted by +) or halving (denoted by -). For example, *MSHR+* is a machine exploiting L1 MSHR tables which have 2X higher MSHR entries than GTX 480 (64 entries per table). Similarly, *MSHRMerger+* is a machine exploiting 2X higher MSHR merger fields per MSHR entry (16 merger fields per entry). The last four groups report performance under machines having 2X higher/lower number of sets/ways in L1 cache.

As reported in the figure, no change to these resources impact int's performance heavily as int performance is limited by memory latency and coalescing stalls. However, smaller data types can take advantage of higher number of merger fields to improve performance. Increasing merger fields by 2X (*MSHRMergers+*) widens the gap between int and char by 26%. In contrary, cutting the number of merger fields to half (*MSHRMergers-*) degrades performance of short and char by 52% and 25%.

The number of mergers impacts performance of short more than char. Under the baseline machine, while char's performance is heavily bounded by the number of mergers, short's performance is not yet limited by the number of mergers. However, under *MSHRMergers-* machine, performance of short and char both are limited by the number of mergers and therefore, short performance drops significantly (compared to baseline machine). In this case, char outperforms short since both are limited by the number of mergers and char has higher memory efficiency.

VIII. Discussion

Different architectures. We have re-evaluated our finding on alternative GPGPUs: NVIDIA GeForce GTX640 and GTX280. Memory hierarchy of GTX640 is similar to our baseline GTX480 and it exploits L1 data cache per SM. Under GTX640, we observed a similar trend to GTX 480. char data type performs the worst in *stereo-matching*, *matrix-matrix multiplication*, and *1D stencil*. short outperforms int in *stereo matching* and *1D stencil*. int data type performs the best under *matrix-matrix multiplication*. However, under GTX280 a differ trend is observed; smaller data types consistently perform better than larger. This is the result of the absence of L1 cache in GTX280, mitigating the impacts of MSHR contention under smaller data types.

Memory pattern. We expect to see high number of stalls on MSHR merger fields in applications exhibiting the following characteristics: i) using small data type for array elements, ii) individual threads of a warp accessing consequent elements of an array, iii) warps of a thread-block accessing consequent data regions. In such applications, smaller data type sizes fit the data of specific thread-block into fewer cache lines. Since the total number of memory accesses does not change with data type variations, these applications experience higher contention on MSHR merger fields under smaller data types.

IX. Related Works

Kerr et al. [11] introduced several metrics to analyze the behavior of CUDA programs. The metrics represent memory intensity, memory efficiency, and branch divergence behavior. Lei et al. [9] implemented the mixed precision algorithm for linear algebra in CUDA. The mixed precision algorithm uses single-precision calculation and precision refinement process to achieve the precision of double, while significantly improving performance. Jia et al. [10] introduced a regression-based model to predict GPGPU workload execution time. They also used the model to identify key bottlenecks of GPGPU microarchitecture. According to their analysis, SIMD width is the most impacting parameter across the evaluated workloads. Jia et al. [12] introduced MRPB to mitigate cache thrashing. They performed evaluation under the cache system similar to what we have studied in this paper. MRPB reorders/bypasses the requests to avoid inter-warp cache contentions. MRPB is a unit inside the LSU reordering the requests after the coalescer module. Although they discuss lack of MSHR entry or cache line stalls, they overlook MSHR merger stalls.

X. Conclusion

Developers of data-parallel applications decide the data type for individual elements based on a tradeoff between performance and calculation precision. In this study, we relaxed the precision constraint and studied the performance impact under GPGPU microarchitectures. First, we showed that the variations of data type size may improve or degrade performance, depending on the application behavior. Then, we investigated the bottlenecks in large/small data type sizes through a cycle-accurate GPGPU simulator. Specifically, we identified key bottlenecks in memory access and miss handling resources. We showed coalescing stalls are critical under larger data types. We also showed miss merging resources are performance limiting under smaller data types.

XI. Acknowledgments

We thank anonymous reviewers for their valuable comments. This work was supported by School of Computer Science at the Institute for Research in Fundamental Sciences (IPM), UVic Fellowship of University of Victoria, and the Natural Sciences and Engineering Research Council of Canada, Discovery Grants Program.

References

[1] L. Nyland et al. Fast n-body simulation with cuda. GPU gems, 3, 677-695. 2007.

[2] T. Aila and S. Laine. Understanding the efficiency of ray traversal on GPUs. In Proceedings of the Conference on High Performance Graphics 2009 (HPG '09), Stephen N. Spencer, David McAllister, Matt Pharr, and Ingo Wald (Eds.). ACM, New York, NY, USA, 145-149.

[3] V. Volkov and J. W. Demmel. Benchmarking GPUs to tune dense linear algebra. In Proceedings of the 2008 ACM/IEEE conference on Supercomputing (SC '08). IEEE Press, Piscataway, NJ, USA, , Article 31 , 11 pages.

[4] P. F. Felzenszwalb and D. P. Huttenlocher. Efficient Belief Propagation for Early Vision. Int. J. Comput. Vision 70, 1 (October 2006), 41-54.

[5] D. P. Playne et al. Numerical Precision and Benchmarking of Very-High-Order Integration of Particle Dynamics on GPU Accelerators. In HR. Arabnia, & MGS. Ashu (Eds.) Proceedings of the 2011 International Conference on Computer Design. (pp. 83 - 89).

[6] A. Bakhoda et al. Analyzing CUDA workloads using a detailed GPU simulator. IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS). pp.163,174, 26-28 April 2009

[7] J. Nickolls et al. Scalable Parallel Programming with CUDA. Queue 6, 2 (March 2008), 40-53.

[8] NVIDIA Corp. Profiler Userś Guide 2013. Available: http://docs.nvidia.com/cuda/profiler-users-guide/

[9] W. Lei et al. Accelerating Linpack Performance with Mixed Precision Algorithm on CPU+GPGPU Heterogeneous Cluster. IEEE 10th International Conference on Computer and Information Technology (CIT). pp.1169,1174, June 29-July 1 2010

[10] W. Jia et al. Stargazer: Automated regression-based GPU design space exploration. IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS). pp.2,13, 1-3 April 2012

[11] A. Kerr et al. A characterization and analysis of PTX kernels. Workload Characterization. IISWC 2009. pp.3,12, 4-6 Oct. 2009

[12] W. Jia et al. MRPB: Memory Request Prioritization for Massively Parallel Processors. The 20th Int. Symp. on High Performance Computer Architecture (HPCA 2014)

[13] H. Wong et al. Demystifying GPU microarchitecture through microbenchmarking. Performance Analysis of Systems & Software (ISPASS), 2010 IEEE International Symposium on , vol., no., pp.235,246, 28-30 March 2010

[14] J. Meng et al. Dynamic warp subdivision for integrated branch and memory divergence tolerance. SIGARCH Comput. Archit. News 38, 3 (June 2010), 235-246.