GPU Bit-Slice Vectors

Kevin Pick
Dept. of Computer Science
University of Texas at Austin
cyberkpick@utexas.edu

Abstract

This paper introduces the concept of bit-slice vectors for unsigned integer storage and addition on the GPU. By deriving unsigned integer formats from bit-slicing principles, we can decrease memory consumption and improve overall execution time for variables of arbitrary precision. We apply our formats to modern GPU programs and generalize our optimizations for further applicability. Our goal is to demonstrate the usefulness of bit-slice vectors on the GPU in order to address the dearth of papers on this topic.

1 Introduction

Despite the advancements in GPU architecture and card construction techniques, two bottlenecks have remained constant throughout the existence of graphics cards: memory access and per thread computation time. The former has consistently increased in minimum latency over each generation of Nvidia architecture from the Tesla GT200 series to that of the Maxwell GM107. With device memory access spiking at over a 340 clock cycle latency on Maxwell cards, memory dependency is clearly an inherent performance challenge for modern GPU algorithms (Andersch, Lucas, Alvarez-Mesa, and Juurlink, 2014).

Similarly, per thread computation time is a vital concern to produce a performant GPU program. Unlike memory access, the problem cannot be resolved by future advancements in GPU design. Since GPU frameworks execute a homogenous set of program instructions on a myriad of cores, any inefficiencies in the program will be replicated to all cores evaluating it. For reference, in the newest card from Nvidia, the RTX 2080 Ti, 4,352 cores can be executed simultaneously to evaluate a workload (Nvidia, 2019). Ergo, it is imperative to optimize a program as much as possible since minor time saves on a single execution will have drastic performance implications.

To reduce the overall dependency on memory as well as increase per thread performance in GPU programs, we introduce multiple bit-slice vector data representations of unsigned integers. At a high level, bit-slice vectors permit computers to both store and operate on data of arbitrary precision. Specifically, we are focused on providing formats that reduce unused data as well as allow for unsigned integer addition and common bitwise operations. We will also propose minor hardware modifications to prevent overflow that can corrupt bit-slice vectors on modern hardware.

To understand the benefits of using our formats for bit-slice vectors, consider the addition of three nine-bit unsigned integers by another set of nine-bit unsigned integers. Normally, a programmer would allocate the closest supported datatype capable of storing each individual element (in this case six 16-bit unsigned integers). Under our model, only two 32-bit unsigned integers would need to be allocated to both store the data and compute the result. Furthermore, while the six 16-bit unsigned integer model requires a total of three ADD instructions on certain GPUs, our model only utilizes a single ADD operation. Hence, in this example, our model not only utilizes 2/3s of the memory cost, but also is significantly faster.

Ultimately, our contribution is utilizing bit-slice vectors for representing low-precision data types on the GPU. Previous GPU papers have not considered how to leverage formats associated with bit-slicing to accelerate computations and compute approximate results when acceptable. Ergo, we will be presenting and evaluating several classes of algorithms for which bit-slice vectors can drastically improve the efficiency of the GPU program.
1.1 Overview
The first half of the paper is focused on introducing the foundation of our contribution to the lay reader. We will discuss common GPU terminology and concepts, describe our proposed bit-slice vector formats, and propose hardware modifications to prevent overflow. The remaining portion of the paper discusses the application of our formats to common GPU algorithms. To compare GPU programs, we utilize a cycle-accurate simulator, GPGPU-Sim 3.2.0 (Bakhoda, Yuan, Fung, Wong, and Aamodt, 2009), to execute CUDA 3.0 kernels on a Tesla C2050. To assess each algorithm, we will compare the overall clock cycles as well as discuss reasons for the differences in cycle count between algorithms.

2 GPU Basics
Before we proceed with the paper, it would be of interest to the average reader to cover basic general purpose GPU architecture and terms associated with graphics cards. Since we are using CUDA, we will defer to Nvidia’s terminology whenever possible.

Figure 1 depicts an overview of a CUDA compliant graphics card. Dissimilar to CPUs which primary function on a Single Instruction Multiple Data (SIMD) model, CUDA utilizes a Single Instruction Multiple Thread (SIMT) architecture. SIMT architectures execute the same instruction on several threads simultaneously. Hence, the standard Nvidia card contains thousands of processors as well as its own memory system to support the parallelism in a SIMT design. Processors in a graphics card are known as multiprocessors (MPs) and have an associated set of stream processors / cores (SP), otherwise known as GPU threads. The overall architecture of a GPU is focused on maximizing parallelism while permitting some form of thread synchronization. Hence, modern GPUs have no concept of device wide synchronization due to the cost of implementation and inefficiency of using such a mechanism. GPU manufacturers instead opt to cluster threads into synchronizable groups known as blocks. Only threads within the same block will wait on fences. To run a GPU program, more commonly referred to as a kernel, a collection of blocks, known as a grid, is executed with all threads running the same program. Because a block must fit within an MP, there exists a maximum number of threads per block. These threads will share the same resources to fetch instructions via the instruction unit (IU), to utilize the special function unit (SFU) for transcendental instructions such as sine, cosine, and reciprocal, and to access shared memory (SM). Shared memory permits SPs to transfer information between each other with relatively low latency. While SPs contain a register based local memory system to store data, the data within shared memory is accessible across the entire block. The final form of memory management in a GPU is known as device memory. Since device memory is located outside of the MP, there is a very high latency associated with it. However, device memory is visible to all MPs and persists between kernel executions. All other forms of memory cannot be accessed after the kernel is terminated. Hence, all kernel inputs and outputs must be written to device memory.

![Figure 1: Basic architecture of an Nvidia GPU. Each acronym is defined as follows: MP (Multi Processor), IU (Instruction Unit), SP (Streaming Processor / Core), SFU (Special Functions Unit), and SM (Shared Memory).](image-url)
3 Bit-Slice Formats

We propose two different formats to represent bit-slice vectors. Unlike previous CPU based papers such as (Xu and Gregg, 2017), our focus is on leveraging existing hardware support whenever possible to accelerate operations. The first format, which we will refer to as transposed fully-packed bit-slice, is friendly to data storage when radix extraction is necessary. The second format, packed bit-slice, is intended to be used when unsigned integer (unit) arithmetic is required.

3.1 Transposed Fully-Packed Bit-Slice Vectors

Array \( a \) of 7-bit Uints Stored in 8-bit Variables

<table>
<thead>
<tr>
<th>( a[0] )</th>
<th>( b_0 )</th>
<th>( b_1 )</th>
<th>( b_2 )</th>
<th>( b_3 )</th>
<th>( b_4 )</th>
<th>( b_5 )</th>
<th>( b_6 )</th>
<th>( b_7 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

* Note: 64-bits are unused.

Transposed Fully-Packed Bit-Slice Vector

<table>
<thead>
<tr>
<th>( b_0 )</th>
<th>( b_1 )</th>
<th>( b_2 )</th>
<th>( b_3 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

* Note: 7-bits are unused.

**Figure 2:** Converting an array of 63 7-bit units stored in 8-bit variables to the transposed fully-packed bit-slice format. Each bin associated with the new format is a length two array of 32-bit uints (holds 64 bits). \( b_i \) represents the data stored at bit index \( i \).

The transposed fully-packed bit-slice vector representation reduces both load time and data waste for applications that only require certain bits of an n-bit uint. For reference, a fully-packed datatype contains no padding except for the last element if necessary. As the name implies, an array of n-bit units is transposed and fully-packed into n-bins. Each bin is an array containing the radix associated with a bit index from the original array.

**Figure 2** shows an example of the process on two 7-bit units.

Since each bit index is stored in a separate bin, operations that swap bits are transformed into lookup tables on bin indices. Furthermore, as no data is lost during the format conversion, the original uint can be reconstructed if necessary.

The largest gain from using the format comes in the form of load efficiency. Instead of reading an entire 32-bit uint to extract a single bit of data, 32-bits from 32 different units can be loaded simultaneously. Practically no unnecessary loads are performed since padding is almost nonexistent for large datasets.

3.2 Packed Bit-Slice Vectors

The packed bit-slice vector format leverages existing hardware whenever possible to compute the resultant value. Each packed bit-slice vector of \( n \)-bits will fit as many variables as possible into a data type of \( k \)-bits. The layout within the \( k \)-bit datatype will be fully-packed. For example, assuming \( n = 10 \) and \( k = 32 \), three 10-bit uints will be packed together inside a single 32-bit variable as **Figure 3** shows.

**Figure 3:** Packed bit-slice vector for 10-bit integers stored in 32-bit variables.

In cases where integer addition does not cause overflow, current hardware is sufficient to compute the resultant value. Under the aforementioned assumption, the sum of two packed bit-slice vectors can be computed by simple addition. **Figure 4** visually demonstrates when addition is successful for pact bit-slice vectors.

As long as dependencies do not exist between units within the packed bit-slice vector, the correct answer is computed with existing hardware. Hence, operators such as \&\&, |, and ^ will always function as expected. However, issues will arise with bit-shifting, multiplication, and division due to dependencies which do not account for the packed bit-slice format.
3.2.1 Hardware Modifications

To prevent addition with overflow from corrupting packed bit-slice vectors, we propose minor hardware modifications to the traditional 32-bit adder. We base our modifications on a carry-adder model in order to estimate the latency caused by the introduction of additional logic gates. Furthermore, we assume our alterations have no effect on standard addition as signal bits passed to the arithmetic logic unit (ALU) can disable the feature.

Figure 5 depicts the altered 32-bit carry-adder which has been renamed to a 32-bit bit-slice adder. Compared to the carry-adder, our model uses an additional 32-bit register for input and contains extra logic to prevent carry bit propagation and overflow. We will refer to the new register from now on as the mask register. To utilize the 32-bit bit-slice adder, the mask register must be informed of the location of each uint within the bit-slice vector. Hence, the corresponding bit-index of the highest bit within each unit is set to 0x1 within the mask register. The two registers that are added together are assumed to be in the same packed bit-slice vector format.

When packed bit-slice vectors A and B are added, any units within the resultant value that overflow will instead be set to their maximum values respectively. Although the uints are added simultaneously, the results can still be thought of in isolation.
4 Algorithm Extensions

To approximate the gains from utilizing bit-slice vectors, we will modify and compare standard GPU kernels to their traditional counterparts. Through the process of altering each algorithm, we will be able to draw generalized conclusions regarding the applicability of bit-slicing on the GPU.

4.1 Standard Radix Sort

Radix sort is widely considered to be the most efficient sorting algorithm for the GPU. Hence, any improvement to radix sort can be considered a worthwhile accomplishment.

The first radix sort algorithm we will consider is utilized for sorting indices associated with Morton Codes. Morton Codes provide a mapping from n-dimensional space to one dimension. More formally, Morton Codes are defined as a space filling curve which has the property of numerical closeness correlating to spatial locality. To visualize the advantages of Morton Codes, we will consider the 2D case in Figure 6.

<table>
<thead>
<tr>
<th>Morton Code</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>(0, 0)</td>
<td>(1, 0)</td>
<td>(2, 0)</td>
<td>(3, 0)</td>
</tr>
<tr>
<td>1</td>
<td>(0, 1)</td>
<td>(1, 1)</td>
<td>(2, 1)</td>
<td>(3, 1)</td>
</tr>
<tr>
<td>2</td>
<td>(0, 2)</td>
<td>(1, 2)</td>
<td>(2, 2)</td>
<td>(3, 2)</td>
</tr>
<tr>
<td>3</td>
<td>(0, 3)</td>
<td>(1, 3)</td>
<td>(2, 3)</td>
<td>(3, 3)</td>
</tr>
</tbody>
</table>

**Figure 6:** 2D Morton Codes. The light blue path demonstrates the ordering of 2D coordinates in Morton Codes starting at Morton Code 0, coordinate (0, 0), and ending with Morton Code 15, coordinate (3, 3).

Notice the Morton Codes wind throughout the 2-dimensional space in a pattern of connected Z’s. For this reason, Morton Codes fill a Z-order curve. Due to the shape of the curve, we can see that the Morton Codes for coordinates (1, 1) and (3, 0) are only separated by a single value. Contrariwise, codes for (0, 0) and (3, 3) have fourteen values between them. Since Morton Codes preserve the locality of n-dimensional points, they are popular for memory layouts where accessing nearby points is expected. For this reason, many GPUs utilize Morton Codes as texture cache indices. Additionally, tree based partitioning algorithms require Morton Codes to order leaf nodes in an efficient manner.

To construct a Morton Code from 3D coordinates, the data is first converted to integers by finding each point’s relative position within the scene’s bounding box and flooring the value. The three uint values for x, y, and z are combined into one uint using a process known as bit twiddling. As Figure 7 shows, the bits are interleaved in a repeating order that can be obtained by swapping bit positions.

**Binary Representation of 3D Points**

- x = 0101
- y = 1100
- z = 0111

**Morton Code Construction**

- x = 0 1 0 1
- y = 1 1 0 0
- z = 0 1 1 1
- code = 010111001101

**Figure 7:** Constructing a 12-bit Morton Code.

To utilize Morton Codes for constructing popular GPU data-structures such as k-d and BVH trees, the codes must be sorted. Thus, it is extremely common for Morton Codes to be used in conjunction with radix sort to obtain the final dataset.

The overall kernel set for constructing and sorting Morton Codes is laid out in Figure 8. As is standard, the Morton Code is constructed from three 10-bit uints, creating a 30-bit value. The process is broken down into two phases: Morton
Code construction and sorting. Since the radix sort kernel set requires thirty executions to obtain the final output, it is traditionally considered to be the bottleneck. Our optimizations will target preparing the data for the radix sort under the consideration that data preparation cost is insignificant. Since radix sort only requires one bit of data per kernel execution, we will utilize the transposed fully-packed format to store the Morton Codes. The Morton Code data will be transposed in matrices of size $32 \times 30$ (Warren, 2012). Because Morton Codes are formed by shifting bits around and our data format allows for access to an arbitrary bit index array, bit-twiddling turns into a lookup table.

<table>
<thead>
<tr>
<th>Morton Codes</th>
</tr>
</thead>
<tbody>
<tr>
<td>- Convert input data to Morton Codes</td>
</tr>
<tr>
<td>- Create initial position indices</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Radix Sort Part I</th>
</tr>
</thead>
<tbody>
<tr>
<td>- Block prefix sum</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Radix Sort Part II</th>
</tr>
</thead>
<tbody>
<tr>
<td>- Global prefix sum</td>
</tr>
<tr>
<td>- Write sorted indices</td>
</tr>
</tbody>
</table>

Repeat for 30 iterations.

Finished

**Figure 8:** Radix Sort kernel overview. Each box with an underlined title is a new kernel execution.

The largest optimization of utilizing the fully-packed format is within the first kernel of radix sort. Normally, each thread loads in one radix per iteration. However, with our modifications, the kernel is capable of loading thirty-two radices per thread. Our optimization also extends to the prefix sum used within the first radix sort kernel. For reference, all versions of radix sort require a prefix sum to compute the new position of data after each iteration. **Figure 9** demonstrates the concept of prefix sum. A parallel prefix sum of a list of size $n$ requires $2 \times \log(n)$ parallel additions with thread synchronization to complete. To reduce the complexity of the operation, we utilize a CUDA intrinsic to compute the sum of bits inside a uint. Because radix sort operates off the prefix sum of input bits, the sum of thirty-two bits is equivalent to the sum of thirty-two radices. The overall operation reduces the complexity of the prefix sum to $2 \times \log(n / 32) = 2 \times \log(n) - 10$. Since blocks can only hold up to 1024 threads, the algorithm takes twenty parallel additions with group synchronization. In our model, only ten of these operations are needed granted the same number of radices are read in. While addition alone is not costly to compute, synchronizing threads within a block is understandably far more expensive. Also, if our proposed kernel were using all 1024 threads, the kernel would output thirty-two times more data due to the difference in load factor. While a regular radix sort kernel would read and write 1024 values, ours does 32768 with the same memory and resource allocation.

**Prefix Sum**

<table>
<thead>
<tr>
<th>Input: 0</th>
<th>1</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output:  0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

**Figure 9:** A prefix sum is a sequence of partial sums such that given an input \{x, y, z, \ldots\}, the output array is \{0, x, x + y, \ldots\}.

Since our model requires data from thirty-two different integers to be packed together, a slowdown is encountered in the second kernel of radix sort. The second kernel is responsible for writing the indices into the correct positions and preparing data for the next iteration of the algorithm. The kernel must write a single bit into an arbitrary index in an output uint as the transposed fully-packed data type requires. Additionally, the kernel is assured each bit-index is only written to once per algorithm iteration due to the way radix sort operates. Hence, multiple blocks can write to the same uint simultaneously. To solve for the issue, the output array is zeroed out prior to the kernel’s execution. Additionally, atomic or is used to write to the output array in order to set a single bit. In the case a zero should be written, the kernel simply skips writing the value as the array was zeroed to begin with. Since atomic operations are considered costly, the
efficiency of the bit-slice radix sort depends on the speedups associated with the first radix sort kernel outweighing the slowdown encountered by the second kernel.

To compare the regular radix sort to the bit-slice radix sort kernel, we uniformly random sample a 1x1x1 cube for points to convert to Morton Codes. As displayed in Figure 10, the bit-slice radix sort kernel becomes more desirable as the reduction in thread count outweighs the usage of atomic operations. In other words, as the input size becomes larger and requires more threads to compute the result, the gains of the bit-slice radix sort kernel increase.

Since the fully-packed bit-slice format was employed, no hardware modifications are required to obtain the proposed kernel. Although our kernel assumed Morton Codes were input into the radix sort, the core optimizations associated with our bit-slice radix sort are extensible to numerous input datatypes. If the GPU produces the dataset being sorted, the radix sort bit-slice kernel set can be used. GPU architecture requires kernels to write data to global memory, meaning the cost to convert the dataset to fully-packed bit-slice is insignificant as the work is done inside of the block. The code would simply be appended to the kernel producing the dataset, meaning no additional global memory access is required. The same amount of data is loaded into the kernel and output, albeit in a different format.

4.2 4-Way Radix Sort

While the former section assumed the GPU processed and modified the input data, this section will not alter input in any manner. Additionally, we will opt to utilize a more memory intensive, yet faster version of radix sort to compute the resultant array. The sort will also operate on the values themselves rather than indices.

Under the 4-Way Radix Sort model (Ha, Kruger, and Silva, 2009), the sort occurs in three stages as shown in Figure 11. The first stage, presort, sorts the data within each block respectively. The second stage performs a massive parallel prefix sum over several kernel executions. The final kernel writes sorted data to memory.

4.2 4-Way Radix Sort

While the former section assumed the GPU processed and modified the input data, this section will not alter input in any manner. Additionally, we will opt to utilize a more memory intensive, yet faster version of radix sort to compute the resultant array. The sort will also operate on the values themselves rather than indices.

Under the 4-Way Radix Sort model (Ha, Kruger, and Silva, 2009), the sort occurs in three stages as shown in Figure 11. The first stage, presort, sorts the data within each block respectively. The second stage performs a massive parallel prefix sum over several kernel executions. The final kernel writes sorted data to memory.

Since the fully-packed bit-slice format was employed, no hardware modifications are required to obtain the proposed kernel. Although our kernel assumed Morton Codes were input into the radix sort, the core optimizations associated with our bit-slice radix sort are extensible to numerous input datatypes. If the GPU produces the dataset being sorted, the radix sort bit-slice kernel set can be used. GPU architecture requires kernels to write data to global memory, meaning the cost to convert the dataset to fully-packed bit-slice is insignificant as the work is done inside of the block. The code would simply be appended to the kernel producing the dataset, meaning no additional global memory access is required. The same amount of data is loaded into the kernel and output, albeit in a different format.

4.2 4-Way Radix Sort

While the former section assumed the GPU processed and modified the input data, this section will not alter input in any manner. Additionally, we will opt to utilize a more memory intensive, yet faster version of radix sort to compute the resultant array. The sort will also operate on the values themselves rather than indices.

Under the 4-Way Radix Sort model (Ha, Kruger, and Silva, 2009), the sort occurs in three stages as shown in Figure 11. The first stage, presort, sorts the data within each block respectively. The second stage performs a massive parallel prefix sum over several kernel executions. The final kernel writes sorted data to memory.

Since the fully-packed bit-slice format was employed, no hardware modifications are required to obtain the proposed kernel. Although our kernel assumed Morton Codes were input into the radix sort, the core optimizations associated with our bit-slice radix sort are extensible to numerous input datatypes. If the GPU produces the dataset being sorted, the radix sort bit-slice kernel set can be used. GPU architecture requires kernels to write data to global memory, meaning the cost to convert the dataset to fully-packed bit-slice is insignificant as the work is done inside of the block. The code would simply be appended to the kernel producing the dataset, meaning no additional global memory access is required. The same amount of data is loaded into the kernel and output, albeit in a different format.

4.2 4-Way Radix Sort

While the former section assumed the GPU processed and modified the input data, this section will not alter input in any manner. Additionally, we will opt to utilize a more memory intensive, yet faster version of radix sort to compute the resultant array. The sort will also operate on the values themselves rather than indices.

Under the 4-Way Radix Sort model (Ha, Kruger, and Silva, 2009), the sort occurs in three stages as shown in Figure 11. The first stage, presort, sorts the data within each block respectively. The second stage performs a massive parallel prefix sum over several kernel executions. The final kernel writes sorted data to memory.

Since the fully-packed bit-slice format was employed, no hardware modifications are required to obtain the proposed kernel. Although our kernel assumed Morton Codes were input into the radix sort, the core optimizations associated with our bit-slice radix sort are extensible to numerous input datatypes. If the GPU produces the dataset being sorted, the radix sort bit-slice kernel set can be used. GPU architecture requires kernels to write data to global memory, meaning the cost to convert the dataset to fully-packed bit-slice is insignificant as the work is done inside of the block. The code would simply be appended to the kernel producing the dataset, meaning no additional global memory access is required. The same amount of data is loaded into the kernel and output, albeit in a different format.

4.2 4-Way Radix Sort

While the former section assumed the GPU processed and modified the input data, this section will not alter input in any manner. Additionally, we will opt to utilize a more memory intensive, yet faster version of radix sort to compute the resultant array. The sort will also operate on the values themselves rather than indices.

Under the 4-Way Radix Sort model (Ha, Kruger, and Silva, 2009), the sort occurs in three stages as shown in Figure 11. The first stage, presort, sorts the data within each block respectively. The second stage performs a massive parallel prefix sum over several kernel executions. The final kernel writes sorted data to memory.

Since the fully-packed bit-slice format was employed, no hardware modifications are required to obtain the proposed kernel. Although our kernel assumed Morton Codes were input into the radix sort, the core optimizations associated with our bit-slice radix sort are extensible to numerous input datatypes. If the GPU produces the dataset being sorted, the radix sort bit-slice kernel set can be used. GPU architecture requires kernels to write data to global memory, meaning the cost to convert the dataset to fully-packed bit-slice is insignificant as the work is done inside of the block. The code would simply be appended to the kernel producing the dataset, meaning no additional global memory access is required. The same amount of data is loaded into the kernel and output, albeit in a different format.
As the name implies, two radices (four values) are sorted by each iteration of the algorithm. The presort step must loop through the two radices four times in order to compute four different prefix sums. Since the number of threads within a block is 128 due to memory limitations, the maximum value of a prefix sum is 127. Since only 7-bits are required to hold the value of the result for each prefix sum, under the packed bit-slice format, a uint32 can store the data. Each prefix sum is independent, meaning all four can be computed at once. Per the definition of packed bit-slice, we can perform four prefix sums simultaneously.

Due to the optimizations applied to the 4-Way Radix Sort algorithm, the bit-slice version is consistently an improvement over the original. Figure 12 demonstrates the cost in cycles for sorting elements under the original kernel and the bit-slice kernel. The graph demonstrates the bit-slice algorithm is not only more efficient than the original kernel, but also grows at a slower rate. At certain input sizes, the bit-slice version even produces more than a 2x speedup.

Since packed bit-slice addition is assured to never overflow in this kernel set, the algorithm can be utilized by current hardware. The input datatype, unlike the previous section, is unmodified. Hence, this version of radix sort can replace any GPU sorting algorithm utilizing uints. The optimizations used are not directly dependent on the radix sort algorithm. Any algorithm utilizing prefix sums and parallel reduction with known maximum precision can implement our tactics. Common GPU algorithms such as photon mapping, Barnes-Hut, and Ambient Occlusion can exploit our optimizations.

4.3 Blur Kernel

Convolution kernels are utilized throughout GPU algorithms ranging from image processing to neural networks. We will consider a specific case of convolution using a 3x3 blur kernel. For every index in a 2D array, the average of the values surrounding and including the index are computed. Figure 13 shows an example of a blur kernel calculation.
For evaluation, we will utilize 10-bit uints in a packed bit-slice format. Traditionally, a blur kernel sums over nine values and then divides. We instead employ the strategy laid out in Figure 14 to accomplish this utilizing packed bit-slice properties. Unlike traditional blur kernels, our kernel computes three values at once as three 10-bit uints are packed into one load. First, as shown in the figure, the kernel loads three packed bit-slice uints to compute the blur kernel value of the middle term. By adding the three values together, the result contains the sum of each column respectively. Because the output’s middle value requires the sum of the entire block, the values for the sum of the left and right column must be added to it. The left and right values of the result require the center column’s sum which is added to the output values. Finally, the six bit-slice vectors needed to compute the left and right values of the result are loaded in and added. Overall, eight additions are required due to optimizations stemming from the packed bit-slice format. Since the standard blur kernel requires nine additions that are two times faster, the modified kernel spends more time in thread than its predecessor. The speedup from the algorithm will instead come from the overall reduction in threads used to compute the resultant value. Due to the simplicity of the kernel, the overhead of the scheduler is expected to be far larger than the work performed by the blur kernel. Hence, the extra time needed for addition within the kernel is negligible in this case.

Due to the nature of bit-slice addition utilizing the proposed hardware, resultant values that exceed 10-bits are clamped to fit within the packed bit-slice representation. For graphics applications, clamping data to 10-bit uint values is normal within the 10-10-10-2 RGBA color format. Hence, the output of our modified kernel is indistinguishable from that of a standard blur kernel for a single channel within that format.

**Figure 13:** Blur kernel calculation for element E.

![Output](A + B + C + D + E + F + G + H + I) / 9

*Note: each row below is a bit-slice vector

**Add Columns from Red Input**

| A | B | C |
| D | E | F |
| G | H | I |

**Shift and Add Columns Along with Remaining Data**

| A + D + G | B + E + H | C + F + I |
| 0 | C + F + I | B + E + H |
| A + D + G | 0 |

| C | 0 | A |
| F | 0 | D |
| I | 0 | G |

| Result |
| Blur = Result / 9 |

**Figure 14:** Bit-slice Blur Kernel calculation for red elements D, E, and F. Each three-value row represents a bit-slice vector.

**Figure 15** displays the cycles for both the regular and bit-slice blur kernel. The input to the blur-kernel is 10-bits and the output is expected to be the same size. The bit-slice version of the kernel far outperforms the traditional algorithm and boasts up to a 3x speedup on the observed input data tested. As the input becomes larger, the gap between the cycles for both algorithms will only increase as the cost for kernel dispatch becomes
higher. Since integer overflow is expected in the bit-slice vector, the proposed hardware modification is required. However, due to the datatype we are writing to, the output from our kernel is indistinguishable from that of the unmodified kernel.

This post processing effect begins with rendering the scene such that the only colors within it are the crepuscular light sources. Thus, all objects not providing crepuscular light are set to black. Then the scene is rendered as usual with the preset materials. Figure 16 shows an example of the difference between a fully rendered scene and the crepuscular light only render. The algorithm then calculates the effect by using a method known as radial sampling. For every pixel in the image, several samples are taken along the imaginary line between the pixel and the light source. Each sample varies in contribution to the resultant color based on exponential decay.

Although the technique is considered faster than physically based models, it is still expensive. To achieve passable results, 100-500 samples are read per thread. Although the cost of a texture read is heavily reduced via texture caching and dedicated image memory, reading hundreds of samples is abnormal for a kernel.

Figure 15: Blur kernel results on a square image.

4.4 Screen Space Crepuscular Rays

Crepuscular rays are beams of sunlight that seem to radiate from the sun’s location in the sky. The phenomena is popular in games due to the dramatic effect the lighting model has on the resultant image. Since the lighting effect is often considered costly to compute, an approximation of it known as screen space crepuscular rays (Nguyen, 2007) is commonly employed. The algorithm attempts to create the lighting effect utilizing only the data captured by the camera.

To improve the performance of the kernel, our optimizations are directly targeted at reducing the cost of a load for the algorithm while producing similar results. First, we will operate on the image using a single channel which can be mapped to the RGB color space. The technique is known as spectral rendering and is more accurate than the RGB model. Since artists are capable of defining the color of the light source, we ask the color to be defined in spectral space from conception. We also

Figure 16: Fully Rendered Scene v.s. Crepuscular Light Only Render.

Figure 17: Baseline v. Bit-Slice at 500 samples per thread.
specify the spectral color must fit within 10-bits. This optimization is not unreasonable since screen space crepuscular rays are computed on a per light basis, meaning the artist can simply change the spectral to RGB map between light sources if a different color palette is desired.

![Crepuscular Calculation](image1.png)  ![Closeup of Stripes in Image](image2.png)

**Figure 18:** Bit-slice algorithm without blur kernel.

Since the algorithm depends on a sufficient number of samples per thread, we chose to uphold this core principle. However, to improve the performance of the kernel, we opted to compute three output values at once. Since each color is 10-bits, we can pack three of them into a uint32 with the packed bit-slice format. When sampling, we choose to treat each bit-slice vector as a single pixel. Hence, the kernel radially samples the image based on the invisible line between the bit-slice vector’s location and the light sources location onscreen. Due to our alterations, texture sampling cannot take advantage of blend modes. For reference, texture sampling blend modes permit surrounding pixels to influence the returned color value. Since images on the GPU are queried via a floating-point x and y position, values in between pixels can be expressed. In these cases, blend modes interpret the contribution of the input coordinate’s fractional elements to the output value. Fortunately, due to the large number of samples, disabling blend modes does not significantly impacts the resultant image. **Figure 17** compares the output from our method to that of the traditional algorithm.

Due to the approximation of radial sampling employed, the output image displays significant stripes as **Figure 18** demonstrates. The stripes appear due to the assumption that angles between consecutive pixels and the light source are approximately the same. To remove the stripes, we utilize the blur kernel from the previous section immediately following the screen space crepuscular calculation. Since the time required for the blur kernel’s execution is far shorter than that of the screen space crepuscular kernel, we utilize a 64-bit packed bit-slice representation of three 21-bit uints. The usage of larger uints was necessary due to a noticeable lack of smoothness in the output gradient utilizing 10-bit values. Hence, the input color values are 10-bits, whereas the output values are 21-bits.

![Crepuscular Results](image3.png)

**Figure 19:** Crepuscular kernel results on a 1024 x 1024 image.

**Figure 19** compares our approach to crepuscular kernels operating on float3 (RGB float channels), single channel float, and single channel uint. Although uint arithmetic is cheaper than floating point arithmetic, the conversion from float to uint causes the kernel to become more expensive than its floating-point counterpart. Our bit-slice version incurs a similar slowdown by first converting the image to packed bit-slice representation; however, the speedup in the crepuscular kernel far outweighs the cost of data conversion. Not only does our kernel outperform those it is compared to, but it also requires less samples to produce a passable output. **Figure 20** demonstrates the advantages of our algorithm over its floating-point counterpart with a low sample
count. Because we execute a blur kernel and assume consecutive pixels are at the same angle from the light source, the bit-slice version of the algorithm will be less prone to creating choppy images due to a low sample count. Since Crepuscular ray algorithms are expected to produce light shafts with smooth gradients, the baseline implementation, as shown in the figure, does not achieve the desired result of the algorithm with a low sample count. However, due to the consequences of our alterations, the bit-slice algorithm is capable of producing smooth gradients under the same conditions.

Since overflow is possible when adding color channels, the algorithm requires our proposed hardware. As the primary speedup associated with the algorithm is in relation to radial sampling, the optimizations are extensible to all kernels employing the technique. The only caveat is that an approximate result must be acceptable for these kernels.

5 Conclusion

In summation, our work demonstrates the necessity of utilizing bit-slice vector representations for low-precision data-types on the GPU. Our results indicate that bit-slice vectors are applicable to a variety of staple GPU kernels that share general properties. The Standard Radix Sort section argues the importance of the transposed fully-packed format when the GPU has full control over data creation and does not require all bits in successive kernels. The 4-Way Radix Sort example demonstrates the importance of utilizing the packed bit-slice format when summation is necessary, and the input dataset is low-precision. Extending upon the association between packed bit-slice and summation, the Blur Kernel we propose proves bit-slicing accelerates the common calculation of convolutions. Lastly, the Screen Space Crepuscular Ray kernel verifies that the format is robust enough to represent approximations of radial sampling. While our work is not an exhaustive list of all usages of bit-slice vectors on the GPU, it does prove the necessity of utilizing such a format within a wide variety of kernels. Despite the generalization of our techniques, the sheer number of cycles saved on common GPU algorithms warrants the usage of the format. Now, when considering the numerous applications of bit-slicing, it is clear the format cannot be ignored in the pursuit of optimization.

6 References

Symposium on Performance Analysis of Systems and Software (ISPASS). Boston, MA.


Hubert Nguyen. 2007. GPU Gems 3. Addison-Wesley Professional.


Michael Andersch, Jan Lucas, Mauricio Alvarez-Mesa, and Ben Juurlink. 2014. Analyzing GPGPU Pipeline Latency. LPGPU.
