Notes

**Abstract**

**Two problems with parallel summation of floating-point numbers on GPUs are loss of precision and non-reproducible results. The precision loss is due to round-off error propagation, and the lack of bit-accurate consistency across platforms and setups can be attributed to the non-associative nature of floating-point addition.**

**To address these problems, we implemented a new method for efficient bit-accurate parallel summation of double-precision numbers on GPUs. This method provides the summation result with full precision, i.e., without round-off error. Thus, it provides the same result on all architectures and execution setups. We see two main uses for this method: (1) algorithms that benefit from extended precision, such as iterative linear solvers (QUDA, AMGX), and (2) where reproducible results are required, such as in cross-platform libraries, for tuning of execution parameters with result validation, etc.**

Deterministic summation of a vector of double-precision numbers with maximum precision

When computing the sum of a vector of double-precision numbers in parallel, the result is non-deterministic mainly because floating point summation is not associative.

Collange et al. suggest a parallel method for a hierarchical accumulation scheme. The lowest level consists of threads that independently store accurate values in registers using the 2Sum method until the sum overflows, and then pass the remainder to the next level. In the worst case, a thread that uses N registers can only read N+1 values before it needs to pass the remainder. The 2nd level is based on fixed-point accumulators. Each accumulator is an array of 64-bit signed digits that partially overlap. (The high bits that overlap the next digit can be considered as carry-save bits.) When these carry bits overflow, the carry is propagated up or down to the next word. The next levels are used for further reduction, until a single value remains.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |

registers

values

carry?

carry?

carry?

carry?

carry?

carry?

carry?

carry?

next level

. . .

Level 1 (per thread)

`

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |

(in shared memory)

values

Level 2 (per thread)

|  |  |  |
| --- | --- | --- |
| S | EXP | MANTISSA |

extract

calc\_digit()

digit

atomic

atomic

atomic

. . .

carry?

carry?

digits cover entire double values range in fixed-point

In the GPU implementation (OpenCL ) of the 2nd level, each warp works on a separate fixed-point accumulator in shared memory (CUDA notation). The size of the accumulator is configurable, and the implementation uses accumulators of size 312B (39 x 64bit). To add a number to the sum, a thread:

(1) reads the value,

(2) extracts the sign, exponent, and mantissa from it,

(3) determines the corresponding digit in the accumulator,

(4) atomically adds the value to the accumulator, and

(5) propagates the carry (two atomic operation per step).

An alternative method for computing the sum in parallel without loss of precision

The limitation of the 2Sum method (Level 1) is that when the numbers that are added by a thread have high dynamic range, i.e., their exponents differ significantly, the registers quickly fill up and the carry propagates to the next level, which uses atomics and has relatively high overhead. This technique is expected to be very efficient for summation of relatively close numbers. Therefore, an alternative approach for summation is to radix-sort the numbers by into bins with similar exponents, and then reduce-by-key the numbers in each bin using 2Sum as the add operator. After the reduction, a short serial pass over the per-bin results computes the final result.

As mentioned in Sylvain’s paper, the dynamic range limitation of the 2Sum method is as follows: when words are used to store the result (expansions of size x), the method works correctly if the dynamic range of the sum is lower[[1]](#footnote-1) than . For accumulated numbers with exponents in the range , the dynamic range of the sum is at the worst case: . We now compute the size of the exponent range per bin .

Suppose that no more than numbers belong to the same bin, and we are using words to store each result. Then, each bin can hold numbers in a range of exponent values. Since the number of exponent bits in a double is 11, only bins are required. If more than numbers belong to the same bin then they can simply be further partitioned into bins of numbers or less.

The main benefit of this approach is that it is based on standard and highly-optimized primitives: sort and reduce-by-key. The method by Collange et al. makes heavy use of atomic instructions, and has a lot of overhead in checking for carry propagation, which may cause divergence and load balancing issues. In contrast, the newly proposed method is expected to have stable performance.

Dynamic range computation table:

|  |  |  |  |
| --- | --- | --- | --- |
| **double words** | **extended mantissa bits** | **max expon bits/bin** | **(given max exp bits/bin)** |
| **2** | **52** | **5** |  |
| **3** | **104** | **6** |  |
| **4** | **156** | **7** |  |
| **5** | **208** | **7** |  |
| **6** | **260** | **8** |  |
| **7** | **312** | **8** |  |
| **8** | **364** | **8** |  |
|  |  |  |  |
| **6** | **260** | **7** |  |

## Summation of per-thread-block bins

Each thread-block produces a set of x-wide accumulators (bins)[[2]](#footnote-2), where each accumulator contains the sum of all the values that were processed by the block and are inside the exponent range of the bin. Next, for each exponent range, we compute the sum of the accumulators that belong to this range. However, the accumulator of the sum may have to be one word wider, because additional may be required when summing NBlocks numbers in the range.

For example, assume the following setup (illustrated on the next page):

* 2-wide accumulators
* Number of input values is (16GB)

For 2-wide accumulators, each thread block can be assigned up to input values, so that it is guaranteed that no accumulator overflows. Therefore, we execute thread blocks. The required width of the higher-level accumulator is maximal when all N numbers fall into the same bin. In this case, bits are required beyond the exponent range and mantissa bits. In total, the required dynamic range of the higher-level accumulator is , so the higher-level accumulator needs to be 3-wide (3=). Note that the lower 39 mantissa bits in the 3rd word are expected to be 0 (40=52\*3-115).

The last step is computing the final sum of the higher-level accumulators. The goal here is to compute the **most significant word** of the accurate sum, rather than the exact sum. Here, we apply serial summation on the higher-level accumulators from lowest to highest into a top-level accumulator. Each accumulator A can extend up to X bits into the exponent range of the next accumulator B, and B can extend up to 52 bits to the range of A. Therefore, A overlaps B by bits at most, so two most significant words need to be added to the next accumulator, and the last word can be dropped.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 0 | 1 | 2 | … | 63 |

|  |
| --- |
| N double-precision items to sum |

Items per thread-block

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |  |  |  |

exponent

|  |  |
| --- | --- |
| sum | rem |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Blk0 | 0 | 1 | 2 | … | 63 |
| Blk1 | 0 | 1 | 2 | … | 63 |
|  | 0 | 1 | 2 | … | 63 |
|  |  |  |  | … |  |
| Last | 0 | 1 | 2 | … | 63 |

bin=2-wide accumulator

rem

compute bin id

and add to bin

**+**

**rem**

**+**

**rem**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 0 | 1 | 2 | … | 63 |

|  |  |  |
| --- | --- | --- |
| sum | rem | rem2 |

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  | sum | rem | rem2 |
|  |  |  |  |  | sum | rem | rem2 |  |
|  |  |  |  | sum | rem | rem2 |  |  |
|  |  |  | sum | rem | rem2 |  |  |  |
|  |  | sum | rem | rem2 |  |  |  |  |
| … | | | | | | | | |
| sum |  |  |  |  |  |  |  |  |

🡨 sum

Final result!

64

1. Actually, by my calculation, the dynamic range needs to be lower than 2^53. However, this doesn’t change the number of radix-sort bits later. [↑](#footnote-ref-1)
2. Assuming that enough thread blocks were lunched so that iterations are not required [↑](#footnote-ref-2)