FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

A Novel Mapping of Arbitrary Precision Integer Operations to the GPU

BACHELOR'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Bachelor of Science

in

Media Informatics and Visual Computing

by

Matthias Gusenbauer
Registration Number 1125577

to the Faculty of Informatics

at the Vienna University of Technology

Advisor: Associate Prof. Dipl.-Ing. Dipl.-Ing. Dr.techn. Michael Wimmer
Assistance: MSc. Dr. techn. Thomas Auzinger

Vienna, 14th September, 2015

Matthias Gusenbauer Michael Wimmer

Erklärung zur Verfassung der Arbeit

Matthias Gusenbauer
Gablenzgasse 64, 1160 Wien

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 14. September 2015

Matthias Gusenbauer

Abstract

With modern processing hardware converging on the physical barrier in terms of transistor size and speed per single core, hardware manufacturers have shifted their focus to improve performance from raw clock power towards parallelization. Solutions to utilize the computation power of GPUs are published and supported by graphics card manufacturers. While there exist solutions for arbitrary precision integer arithmetics on the CPU there has been little adoption of these libraries to the GPU. This thesis presents an approach to map arbitrary precision integer operations to single threads on the GPU. This novel computation mapping technique is benchmarked and compared to a library that runs these computations on the CPU. Furthermore the novel parallelization technique is compared to an alternative mapping scheme proposed by Langer et al [Lan15]. It is shown that mapping computations to single threads outperforms both the CPU and the approach by Langer. This thesis also explored the feasibility of rational number operations on the GPU and shows that this is in fact practically usable by providing benchmarks.

Contents

CHAPTER 1

Introduction

1.1 Motivation

In this thesis the concept of massive parallelization is applied to computations on integers of arbitrary length. To understand the limitations imposed by using hardware in a non parallelized manner the history of computing power and its current development is explained in the following sections.

1.1.1 History of Computing Power

A long time Moore's Law has been accepted as a reality for the growing number of transistors per CPU die [Moo98]. In 2015 Intel's CEO Brian Krzanich had to admit that Intel can not keep up with Moore's law because the size of the die structures has become so small that the manufacturers have trouble producing CPU dies where the number of rejects is low [SA 15]. The consequence of reaching the limits of the current production process using lithography is that single chip performance will increase more slowly than it has in the past until the natural boundary for minimization has been reached. Manufacturers looked for alternatives and found a solution by duplicating processing cores on a die. While IBM already introduced a multicore processor in 2001 [TDF+02], the multicore architecture gained public interest when it was available as x86 processor for the consumer market in 2005 when Intel as well as AMD introduced their Intel Pentium D and the AMD Athlon 64 X2 [Gee05]. The number of parallel execution units per die has increased since then and it is not uncommon to find hexa-core systems in consumer PCs. To improve performance even further Intel introduced hyper threading which allows the operating system to address two virtual cores per hardware CPU core. AMD delivered a similar approach called Bulldozer themselves. One can see that the manufacturers have acknowledged the problem with exponential shrinking of hardware and are looking for solutions by using parallelization.
Other hardware that can be used to run parallel tasks are the Graphics processing unit

(GPU). The GPU has only been used for displaying visually pleasing images on the computer screen in the past. A GPU has a number of cores that fulfil a special purpose like rasterization of 3D primitives such as triangles. In the past the functionality of these processors was engraved by hardware designs. When programmable shaders emerged people started to creatively use the processing power of GPUs by encoding their data into pixel values. This was only the first step towards the general utilization of the GPU's computational power [OHL$^+$08].

1.2 General-Purpose Computing on Graphics Processing Units (GPGPU)

When hardware vendors realised that there is a necessity for parallelization and that the GPU's hardware design is inherently supporting this, the hardware was opened to software developers to use the processing power of GPUs for more than rendering. The GPU manufacturers started to allow developers to program the hardware as they needed it. This was the birth of General-Purpose Computing on Graphics Processing Units (GPGPU). The manufacturers of graphics cards started to provide the necessary tools and programming languages for developers to use the massively parallel architecture of GPUs [L$^+$04].
While there are many established libraries and tools for running algorithms on the CPU, the diversity of implementations for the GPU is still not comparable to the CPU landscape. This stems from the fact that GPGPU is a relatively young field and the single instruction multiple data (SIMD) architecture does not make it suitable for all problems. There are however cases where transferring the work to the GPU yields speed improvements over performing the same workload on the CPU.
The architectural difference of CPU and GPU cores is their execution paradigm. CPU cores are very sophisticated and follow the multiple instruction multiple data (MIMD) paradigm. This approach allows for completely independent execution of instructions within each core. Hence the name multiple instructions. Although MIMD provides advantages such as multi threading of truly independent execution paths of programmes this freedom comes with the cost of complexity. This complexity manifests in the increased number of transistors per core which means increased physical size on the CPU die.
In contrast to CPU cores GPU cores are comparably small. This reduction in size allows to pack many more cores on a processor die. While commodity CPUs provide the user with six cores, commodity GPUs possess around 2000 to 3000 cores. The increased number of cores is due to the fact that GPU cores are much simpler than CPU cores. On a GPU the cores can not execute completely independent of each other. GPU cores follow the single instruction multiple data (SIMD) paradigm where a number of cores executes the same instruction on a different set of data. However not all cores on a GPU execute the same instruction. On the instruction level cores or threads are grouped together where each thread within the group executes the same instruction, hence the name single instruction multiple data.

1.2.1 Nvidia CUDA

Compute Unified Device Architecture (CUDA) is Nvidia's proprietary solution to provide developers and scientists with a GPGPU framework. When Nvidia published the first version of CUDA in June 2008 it supported Windows and Linux as well as all Nvidia GPUs from the G8x series and onwards. In 2014 Nvidia released the first GPUs with compute capability 5.0 and is currently at 5.3 with the release of their Tegra X1 chip in 2015. The compute capability version gives developers an indication of properties important to execution on the GPU independent of the underlying hardware. Such parameters and features can be maximum grid size, maximum number of instructions per program or even warp size.
At a hardware level threads are executed in parallel on streaming multi-processors (SM). A GPU has several SM to be able to execute a massive number of threads in parallel. Each of these SMs has its own registers, L1 cache and shared memory. Threads are executed in groups of 32 and this group is called a *warp*. Each thread within a warp performs the same instruction on its own set of data. The hardware level execution of threads can be seen in Figure 1.1.

On a logical level threads are grouped into a block which compose the grid. This logical architecture can have up to three dimensions which make it suitable for problems that can be interpreted as a planar structure such as images or even three dimensional space for physics simulations. As threads execute in groups of 32 it is important for performance to find a suitable mapping of warps to the grid. This grouping of threads into blocks as well as the grid layout can be seen in Figure 1.2.

1.2.2 OpenCL

OpenCL is the open source counterpart to Nvidia's CUDA. OpenCL was invented by Apple in 2009 and is now administrated by a consortium called Khronos Group. In this consortium several stakeholders, such as AMD, Apple, Intel, Nvidia and many more are involved [Khr15a]. In contrast to CUDA OpenCL aims not to be a pure GPGPU solution but to provide a heterogeneous platform for running computations on a number of devices. This has the advantage that code has to be written only once and can be run on devices such as the CPU, GPU, digital signal processors (DSPs) and field programmable gate arrays (FPGAs). While it is convenient to have a single codebase to run on many different devices, to achieve optimal performance developers have to adapt the code for each device [Khr15b].

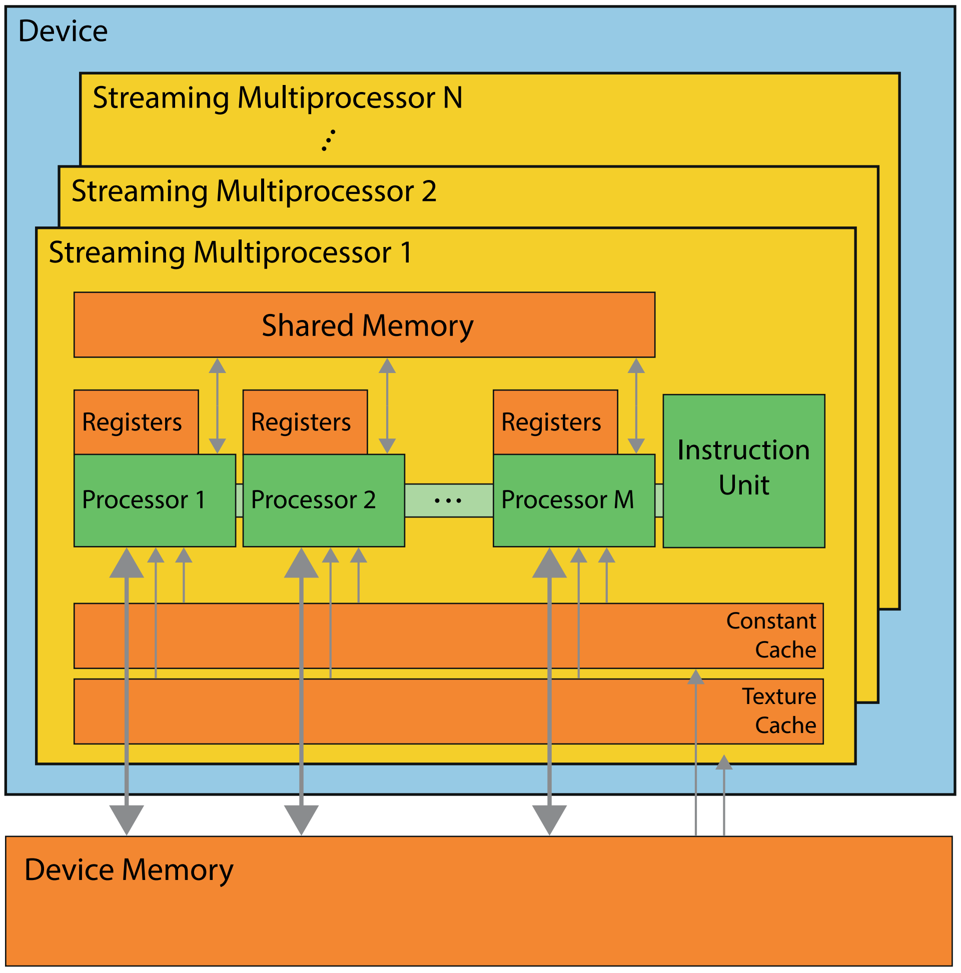

Figure 1.1: CUDA thread execution on a hardware level. It shows the multiple streaming multi-processors and their cores, registers and shared memory. Additionally the global memory which is accessible by all cores but has worse access times can be seen. [Nvi15e]

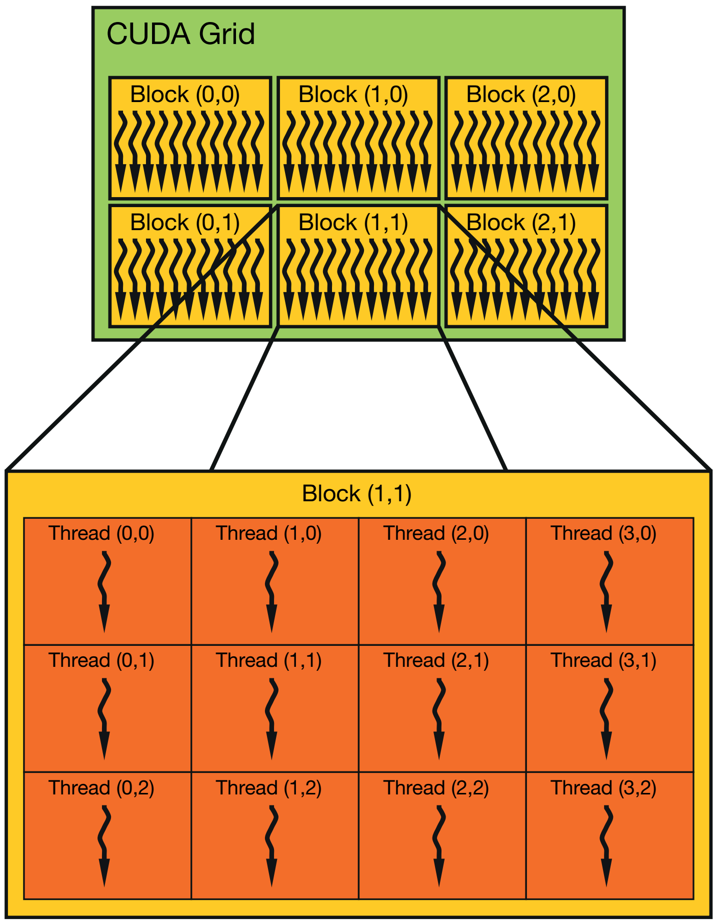

Figure 1.2: CUDA thread grouping on a logical level. One can see the grouping of threads into a block and of blocks into a grid [Nvi15e].

CHAPTER 2

Related Work

2.0.1 Existing GPGPU Implementations

While there are many established tools and libraries that provide CPU support, the landscape for GPU oriented solutions is not as developed. This stems from the fact that GPGPU is a relatively young field and not all algorithms can be adapted to perform well in the SIMD execution model. There are projects that leverage the GPUs computing power to solve specialized problems such as password cracking [pyr15], machine learning [JSD^{+}14] [DYDA12], networking [HJPM10] [VAP^{+}08] or computer vision [WACS11] [CMS12]. Apart from using the GPUs computational power for specialized fields there are attempts to provide others with tools that can be applied to more general problems. One such step is the publication of extended precision number formats on the GPU. In 2010 Lu et al. published a paper on extended precision on graphics processing units that supported floating point numbers [LHL10].
However there are important tasks that require *integers* with higher precision than supported by processors. One such example would be public key cryptography which requires large prime numbers that can not be held by a single 64 bit integer. When performing a multitude of such computations, running the operations on the CPU becomes infeasible. GPGPU with massive parallelization outperforms the CPU if there is a large number of operations and algorithms are suitable for the SIMD paradigm [LHL10].

2.1 GPGPU Libraries

With the advent of GPGPU and manufacturers giving developers and scientists access to the GPUs computing power by introducing CUDA and OpenCL, developers started to create tools and libraries to move workload from the CPU to the GPU. This led to the implementation of many specialized GPU solutions that match a broad set of requirements. One of the most well known CUDA libraries is Thrust which provides the developer with tools for fast parallel algorithms and data structures [nvi15d]. Other noteworthy solutions are cuBLAS [nvi15a], a GPU implementation for basic linear algebra subprograms, and cuFFT to perform fast fourier transformation on the GPU [nvi15b]. This is a small excerpt of libraries written using CUDA. A list compiled of implementations using CUDA can be found on the Nvidia CUDA Zone [nvi15c] homepage. AMD also provides an overview of libraries written in OpenCL on their OpenCL developer zone [amd15] website.

2.1.1 Multi- and Arbitrary Precision

Apart from the libraries mentioned in Section 2.1 there have been implementations to support number formats with higher or even arbitrary precision numbers compared to the standard 64 bit and less number representations. In 2010 Lu et al [LHL10] published their work on extended precision floating point numbers. They designed a solution that includes two different precision modes based on 64 bit floats, arithmetic operators and mathematical functions. While one of the increased precision modes is limited to floats of up to 212 bits the second mode supports arbitrary precision but suffers from decreased performance when compared to the former.
In 2014 Valentina Popescu published her work called "CudA Multiple Precision ARithmetic librarY" (CAMPARY) which also works on floats with increased precision [PJ14]. While her work supports arbitrary precision and arithmetic operations the computations of these operations themselves are not parallelized. In addition to her work Popescu published benchmarks of her implementation for addition, multiplication, reciprocal and division that measure the performance of her work.
Langer et al [Lan15] published parallelization on the GPU on an computation level. His work is performing arithmetic operations on integers of arbitrary length using 32 bit *unsigned integers* as a building block. The same multi-word approach to achieve arbitrary precision for integers is used in this work and is therefore explained in Section 4.2.1 in detail.

2.2 Motivation and Objective of this Thesis

As described in the introduction of this thesis there is a trend towards parallelization. For that reason this work explores additional possibilities to leverage the GPUs computation capabilities. To achieve this the general concept of parallelizing computations as proposed by Langer was adapted to measure the performance of a different parallelization approach. As Langer's work is limited in terms of performance for small integers of arbitrary length a technique is proposed that can be used if the workload is enough for the GPU in terms of numbers of computations and the computations are performed on such small numbers. Additionally the foundation for higher arbitrary precision number representations, such as rational numbers, on the GPU is laid. As parallelization is becoming more important the capabilities of the graphics processor by implementing a different parallelization approach as well as the rational number format is explored.

CHAPTER 3

Algorithms

This chapter describes the novel approach to map computations to threads as well as the necessary algorithms to minimize the impact of memory transfers and mitigate problems occurring by the behaviour of the GPU. In order to be able to understand the necessary steps before actual computation happens on the GPU the algorithms of the computation assignment and the prefix sum are explained in this chapter. However before that the concept of warp shuffles is explained briefly as it is an implementation detail that further boost performance and was used to implement the the computation assignment and the prefix sum.

3.1 Computation Mapping Technique

The computation assignment as described below in Section 3.3 maps one computation to one thread within a warp. To achieve good performance it is important that threads within a warp do not need to wait for data and that the operands of each operation are approximately the same length. Otherwise threads would stall each other because of threads that finished their computation before others and therefor are idle. Another important part is the computation assignment itself. Using atomic operations for computation assignment and memory reservation in each thread can cause a performance bottleneck because large numbers of atomic operations result in almost serial code execution. The following algorithms describe the novel method of computation assignment and memory reservation in global memory for computation-to-thread mapping to mitigate the problematic of atomics.

3.2 Warp Shuffles

Warp Shuffles, as introduced with compute capability 3.0, allow for inter thread and intra warp communication without the need to use shared memory. The advantage of warp shuffles compared to using shared memory is an increase in performance and using less shared memory that is then free to be used by other functions in the same kernel launch. CUDA provides four different types of warp shuffles for integers and floats. There are shuffles that copy a value from a given thread or shuffling a value up or down relative to the current thread. The CUDA C Programming guide [Nvi15e] offers an in-depth documentation of this operation.

3.3 Computation Assignment

Computations are assigned with a computation counter that is saving the next free computation. This counter is used that warps that finished the work previously assigned to them can retrieve the next free computation and start working again. Therefore it is necessary to increase this counter by writing the index of the next free computation to it. As the GPU is inherently parallel a correct memory access pattern has to be guaranteed. To avoid race conditions the computation counter would need to be increased using an atomic operation. This however results in serial execution on the GPU. When few threads are performing atomics and the execution time for operations differ, the chance that threads have to wait for other threads to finish their memory transaction is very low. For our novel mapping technique where each thread would perform an atomic operation the opposite is the case and threads would stall. This serial execution of part of the code would result in bad performance by cancelling out the parallel nature of the GPU.
To mitigate the problem of atomics the computations are reserved warp-wise. The first thread in each warp reserves warp width, which is 32 on current Nvidia GPU's, computations by performing a single atomic operation which increases the computation counter by 32. In the next step each thread in the warp retrieves the beginning index of this computation block by doing a warp shuffle and then each thread adds its thread index within the warp to calculate the corresponding computation index. This approach ensures that only one instead of 32 atomic operations per warp is necessary to perform computation assignment.
Another problem that can occur is that it is not certain how the GPU saturates it's warps. This can lead to warps where not all 32 threads are performing computations. Therefore performance is lost due to the fact that in total there have to be more warps that are not fully saturated with computations. By doing computation assignment as described in Algorithm 3.1 it is ensured that warps are fully saturated with computations and this decrease in performance is avoided.

3.4 Prefix Sum

Another part of the general integer operation algorithm that suffers the same issue as computation assignment is the result memory reservation in global memory. The first naive implementation involved such a solution which lead to massive serialisation of global memory accesses by using atomic operations. After profiling the CUDA application it was clear that using atomics in each thread would pose a memory performance bottleneck.

Algorithm 3.1: Parallel assignment of computation to threads

Input: An enumerated set of computations and an index C_l of the first computation that was *not* issued for computation as well as the total count C_N of available computation. An index that enumerates all threads that are executed in parallel. A warp size W, for which groups of W many consecutive threads belong to the same warp and can share information via a `GetValue(`*value*, *thread index*`)` function.

Output: A computation index c_i for each thread i.

```
i ← GetThreadIndex();            // Initialize local thread index
while $C_l < C_N$ do
    $o_i$ ← ($i$ mod $W$);           // Offset from first thread in warp
    if $o_i = 0$ then
        $c_i$ ← AtomicAdd($C_l$, $W$);   // First thread in warp reserves
    end
    $c_i$ ← GetValue($c_i$, $i - o_i$);  // Propagate value to other threads
    $c_i$ ← $c_i + o_i$;                             // Add local offset
end
```

To improve performance by avoiding shared memory warp shuffles were used for inner warp communication. The goal was to reduce the atomic operations from 32 per warp to only 1 per warp. The idea is that one thread in the warp is aware of the total memory needed to perform all 32 operations within the warp by performing a prefix sum. This thread then reserves the necessary memory chunk in global memory.
By using a prefix sum it was not only possible to sum up the necessary amount of memory but also to ensure that each thread knows where in the reserved memory chunk it should write its results to. A prefix sum for a sequence of numbers $x_0,\ x_1,\ x_2,\ ...$ is another sequence of numbers $y_0,\ y_1,\ y_2,\ ...$ which is computed by the recursive formula $y_i = y_{i-1} + x_i$. An example of such a computation can be seen in Table 3.1.

values	1	5	3	7	2	9	5
prefix sum	1	6	9	16	18	27	32

Table 3.1: Prefix Sum Example

An easy implementation would be to iterate over all threads in the warp and shuffle the maximum length of the result to the next thread. In each of these iterations one has to compute the sum of the previous value and the maximum result length of the current thread. Such an implementation would result in $O(N)$ iterations, given that N is the number of threads per warp. This method can be adapted to decrease the number of iterations and therefore reduce the runtime necessary to calculate the size of the global memory chunk and the positions per thread within this chunk. Once again parallelisation can be used to reduce the number of iterations to $log_2(N)$. Instead of adding the value

Algorithm 3.2: Prefix Sum using Warp Shuffles

Input: An offset counter I marking the first position of free memory in global memory and the size of the result r_i of the computation of thread i. A function `GetValue(`$value, thread\ index$`)` to share a value with a thread within the warp. `GetThreadIndex()` to get the ID of the current thread. `GetLastActiveIndex()` to get the highest index of all active threads. `AtomicAdd(`old_value, add_value`)` to add a value to another value.

Output: The updated offset counter m_i pointing to the memory position where each Thread i writes its result. Furthermore the updated offset counter I for the next free memory chunk.

```
1  $i \leftarrow$ GetThreadIndex();          // Initialize local thread index
2  $l \leftarrow$ GetLastActiveIndex();      // Last active thread id in warp
3  $n_i \leftarrow r_i$;              // Initialize value of maximal result size
4  for $l \leftarrow 1; l < 32; l \leftarrow 2l$ do
5      $n_{i-1} \leftarrow$ GetValue($n_i, i+l$);        // Get value from thread $i+l$
6      if $i \geq l$ then
7          $n_i \leftarrow n_i + n_{i-1}$;          // Sum up values of thread $i$ and $i-1$
8      end
9  end
10 if $i = l$ then
11     $m_S \leftarrow$ AtomicAdd($I, n_i$);       // Increase result memory offset
12 end
13 $m_S \leftarrow$ GetValue($m_S, l$);     // Get memory start position for warp
14 $m_i \leftarrow m_S + (n_i - r_i)$;     // Get start position in reserved memory
```

one thread after another, each thread shuffles its maximum result length to the next thread. In the next iteration each thread adds the value from the previous shuffle to his own maximum length and then shuffles this value to the thread 2 positions ahead. In the next iteration the same happens but the added value is shuffled 4 positions ahead. By doubling the next position to shuffle the value to the number of iterations is reduced from N to $O(log_2(N))$. A schematic visualisation of this thread shuffling behaviour can be seen in Figure 3.1.

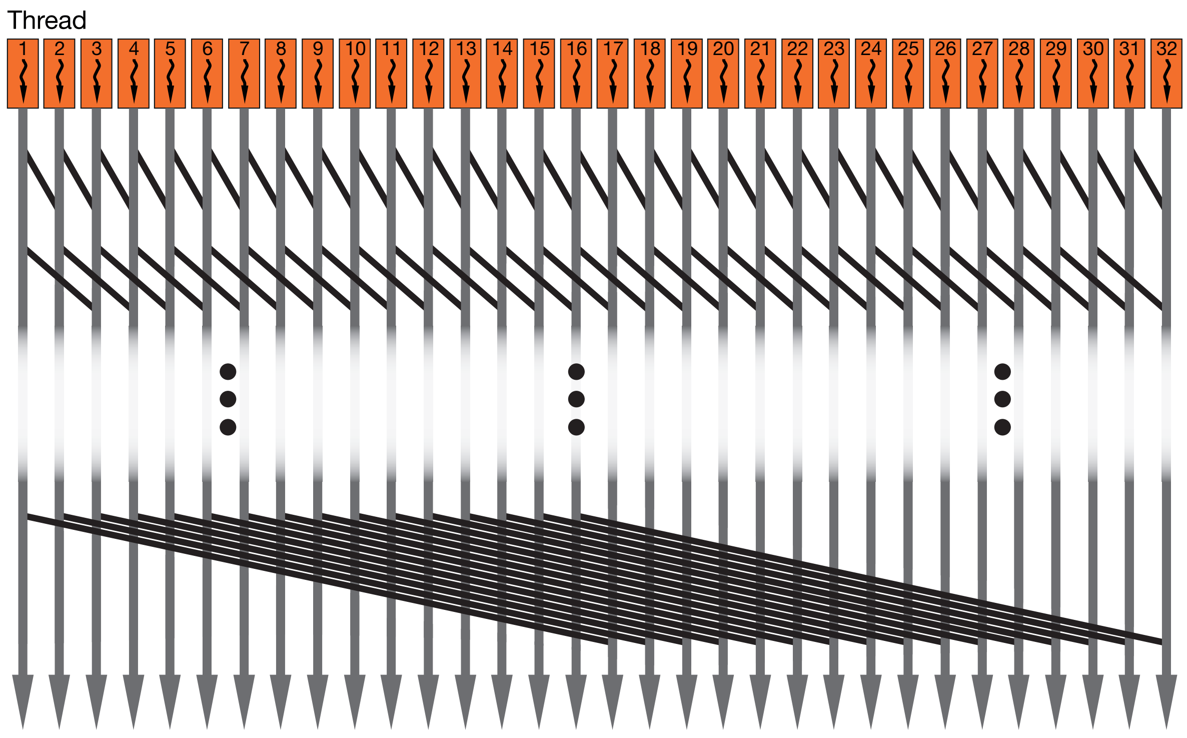

Figure 3.1: The schematic for performing a parallel prefix sum using warp shuffles. The diagonal lines signal when the value n_{i-1} (see Algorithm 3.2) summed up by the thread with a lower ID is passed to the thread with the higher ID. Only the first two and the last warp shuffle and therefore loop iteration are shown to keep this Figure simple.

CHAPTER 4

Methodology

The following sections describe the novel mapping technique presented in this thesis that performs well for computations on integer numbers of arbitrary precision where the size of the operands is equal. To be able to compare this approach to related work in the field, the algorithms behind the novel computation mapping and the arithmetic operations are explained. As this computation mapping is compared to the Langers parallelization technique in Chapter 5 extensively we will explain the basic principle of Langers computation mapping towards the end of this chapter in Section 4.5. First we explain why we chose CUDA over OpenCL although OpenCL is an open standard and can be run on more devices than CUDA.

4.1 CUDA over OpenCL

The decision against the usage of OpenCL was made based on the fact that the developer tools did not support the full feature set necessary for implementing our approach and development would have been cumbersome. When the work on the implementation started in 2014 Nvidia only supported OpenCL version 1.1 on their GPUs which was missing features that are important for optimal performance. On October 17, 2014 the Khronos group published the specification for OpenCL 2.0 which supports some features that are necessary for an optimal implementation of this library. However even with the specification of OpenCL 2.0 the standard does not support warp shuffles. Therefore if the implementation is to be ported to OpenCL in the future, adaptions to the algorithms presented in Chapter 3 have to be made.

4.2 Data Representation

4.2.1 Memory Representation

In order to minimize memory access during the computations the library avoids complex memory constructs which would require multiple lookups in the GPUs memory. This is done because accessing global memory of a GPU is expensive. The reason is the parallel nature of streaming processors. Simultaneous memory accesses by many computing threads may stall the processors because memory accesses are blocking and threads may have to wait for data. This leads to a low occupancy of threads which means that threads are not performing work but idle waiting for data. Therefor all the integers of arbitrary length are saved in one block of global memory in a continuous manner and are accessed by a single lookup using the information saved in the computations. The same continuous memory approach applies for the computation memory.

Integer Representation

An integer of arbitrary length is represented by words. This allows for integers beyond the limit of 64 bit. Each number consists of a number of words necessary to represent the integer of arbitrary length. Each of this words is a *32 bit unsigned integer*. Before starting the computation the total number of necessary words to perform the computation, including the result, is calculated and the necessary memory is allocated and then transferred to the GPU. This memory itself does not hold any information where the big integers are located or where they start and end. This information is only known to the computations themselves.

Computation Representation

A computation in this library is a *struct* which holds the information on where the necessary integers are in memory and which type of computation it represents. The information for an integer is its offset to the beginning of the global memory block and its size. During the computation the correct word for the calculation of position k can be retrieved by accessing the memory at location $offset + k$.

4.3 Computation-to-Thread (CTT) Mapping

Mapping computations to threads comes with the advantage that this approach performs well for integer operations with operands of equal length. Furthermore by mapping one computation to one thread the lower limit of computation time for integers with less than 32 words is avoided. Because each thread performs a computation in a loop and by ensuring that each thread within a warp has a computation assigned all warps are fully saturated independent of the number of words per operand. This means that one warp performs 32 computations in parallel. This decoupling of warps and computations is the reason for increased performance on integers below the size of 32 words or 1024 bits as can be seen in Section 5.3. However as there are no free lunches this advantage comes with the disadvantage that the operands of the computations should be of approximately the same size within a warp. This stems from the fact that if the operands vary greatly within a warp there will be threads that need significantly longer than other threads and therefore stall the streaming multiprocessor. This leads to bad hardware utilization and therefore to decreased performance. The novel mapping method of one computation to one thread can be seen schematically in Figure 4.1.

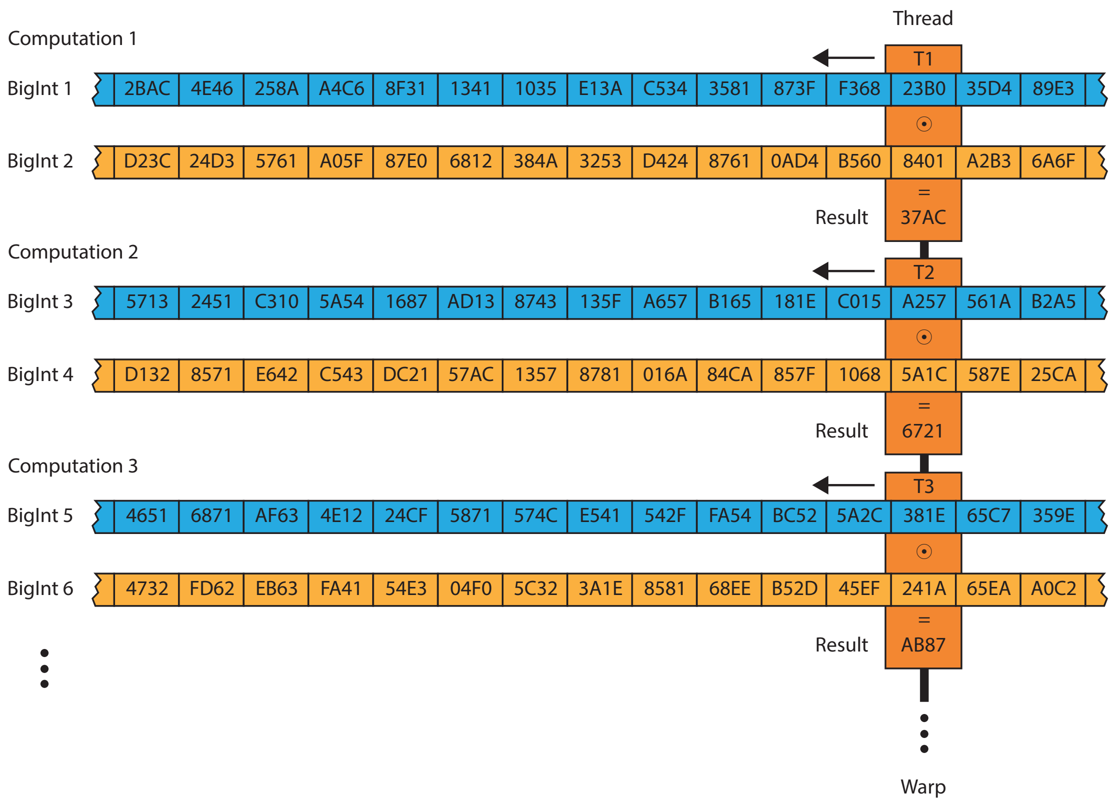
Thread
Computation 1
T1
BigInt 1
2BAC 4E46 258A A4C6 8F31 1341 1035 E13A C534 3581 873F F368 23B0 35D4 89E3
⊙
BigInt 2
D23C 24D3 5761 A05F 87E0 6812 384A 3253 D424 8761 0AD4 B560 8401 A2B3 6A6F
=
Result 37AC
Computation 2
T2
BigInt 3
5713 2451 C310 5A54 1687 AD13 8743 135F A657 B165 181E C015 A257 561A B2A5
⊙
BigInt 4
D132 8571 E642 C543 DC21 57AC 1357 8781 016A 84CA 857F 1068 5A1C 587E 25CA
=
Result 6721
Computation 3
T3
BigInt 5
4651 6871 AF63 4E12 24CF 5871 574C E541 542F FA54 BC52 5A2C 381E 65C7 359E
⊙
BigInt 6
4732 FD62 EB63 FA41 54E3 04F0 5C32 3A1E 8581 68EE B52D 45EF 241A 65EA A0C2
=
Result AB87
Warp

Figure 4.1: A schematic representation of CTT mapping. It shows multiple threads of a warp performing multiple operations on different integers word by word.

4.3.1 Operations

The following sections give insight in how the basic arithmetic operations on integers of arbitrary length are mapped to single threads. Whereas the addition and subtraction have one loop iterating over all words. The multiplication is more complex and two nested loops are necessary. To explain the algorithms in detail let the first integer be denoted as I_1 and the second as I_2. A word at position i in the integer will be $w_{1,i}$ for a word of I_1 and $w_{2,i}$ for one of I_2. int_{max} will denote the maximum value of a 32 bit unsigned integer. The carry generated by an operation of the i-th words of the integers is c_i. For the addition and the subtraction the i-th word corresponds to the i-th loop iteration. The multiplication needs additional notation because it involves two nested loops. This additional notation is explained in the corresponding Section 4.3.1.

Addition

The addition uses the basic school method that iterates over each word of the integers of the operands and adds them together. In each loop iteration detection of carry propagation is necessary due to a possible integer overflow. An overflow is detected by checking if

- $w_{1,i} > (int_{max}$ - $w_{2,i})$
- **or** $(w_{1,i} == int_{max}$ xor $w_{2,i} == int_{max})$ and $c_{i-1} == 1$
- **or** $w_{1,i} == (int_{max}$ - $w_{2,i})$ and $c_{i-1} == 1$

If such an overflow is detected the carry is propagated to the next loop iteration and then added to the result of $w_{1,i+1} + w_{2,i+1}$. This loop has max(length(I_1), length(I_2)) iterations and a maximum result length of max(length(I_1), length(I_2)) + 1 because the last addition can propagate one additional carry at most.

Subtraction

The subtraction of two integers of arbitrary length follows the same principle as the addition with a different approach for carry propagation. Instead of overflowing, subtractions can cause an underflow. An integer underflow and therefore carry propagation occurs if one of the two following conditions is met.

- $w_{1,i} > w_{2,i}$
- **or** $(w_{1,i} == w_{2,i})$ and $c_{i-1} == 1$

Once again the carry c_i is subtracted from the result of x_{i+1} - y_{i+1}. For subtractions the maximum loop iteration count and the maximum result size is the maximum length of the operands.

Multiplication

Multiplication for two integers of arbitrary length is implemented as the standard multiplication algorithm. We are using two nested for-loops where the result r_i at integer word position i is written directly to the result memory. As this parallelization approach is targeted towards integers of arbitrary length with a small word count the runtime of $O(n^2)$ for nested loops poses no problem. Benchmarks showing that this is in fact the

case can be found in Section 5.1.
Using the school method a multiplication is performed by looping over both operands. The outer loop iterates over I_1. For each iteration i the i-th word $w_{1,i}$ is selected. The inner loop iterates over I_2 and multiplies each word $w_{2,j}$, for the j-th iteration of the inner loop, with $w_{1,i}$. The next step is to add the carry of the previous iteration and the result of previous multiplications to the result and save it to memory. For each iteration i of the outer loop the position of the intermediary result is shifted i positions. This shifting behaviour can be seen more clearly in the schematic depiction of the method in Figure 4.3.
In contrast to addition and subtraction, the carry of a multiplication of two 32 bit integers can be 0 or in the interval $[max,\ int_{max}^2 - int_{max}]$. Therefore a different approach for carry propagation is used. Instead of checking for an overflow for each intermediary multiplication of words, the carry is computed and added in the next loop iteration. This is done using the 64 bit capabilities of GPUs for storing the intermediary result of a computation. The 64 bit features are leveraged by writing the result of $w_{1,i} \times w_{2,i} + c_{i-i}$ into a 64 bit integer. Then the result r_i is assigned by using a bit-mask to only select the lower 32 bits of the 64 bit result. The carry is calculated by shifting the 64 bit result 32 positions to the right. A complete pseudo-code description of the algorithm showing the nested loops, position shifting and memory access can be found in the appendix in Section B.

4.4 Rational Numbers

This section provides an overview on the implementation of rational numbers within the library. At first the ambiguous memory representation of rational numbers as well as the reason for choosing such a representation are explained. The second part of the section is explaining the four basic arithmetic operations on rational numbers and the concept behind them. A detailed explanation of the basic operations used for rational number computations can be found in Section 4.3.1.

4.4.1 Memory Representation

Rational numbers in memory are similar to the simple representation of integers. All the information on rational numbers is saved within a rational number computation struct. This struct holds the information on where in the global memory of the GPU the numerator and the denominator are located. The rational computation struct holds 12 values where 4 values are reserved for one rational number. For each rational number the numerator size and offset in memory as well as the size and offset of the denominator are saved. Although this representation of rational numbers is not intuitive it offers great GPU memory look-up performance. Complex data structures where nominator, denominator and rational number have their own representation would lead to cascading memory look-ups. To find the actual number in memory a lookup in the memory sections of these datatypes would have to be performed. Instead of accessing the offset and size in the computation struct directly 3 additional memory look-ups to retrieve the rational number, the nominator and the denominator would have to be performed. These additional look-ups can be avoided by using memory access optimized data structures.

4.4.2 Operations

With the existing implementation of operation for integers of arbitrary length the extension to provide the same functionality for rational numbers was trivial. Consider the two rational numbers $F_1 = \frac{a}{b}$, $F_2 = \frac{c}{d}$ and the result number $R = \frac{x}{y}$. An addition of F_1 and F_2 can be broken down into three multiplications and one addition. The following sections explain how the basic arithmetic operations of integers with arbitrary length have been extended to work with rational numbers.

Addition

An addition of two rational numbers F_1 and F_2 can be written as $\frac{a}{b} + \frac{c}{d} = \frac{(a\times d)+(c\times b)}{b\times d} = R$. Because these operations are integer operations the existing implementation for computations of integers are used. The library creates integer computations and calls the multiplication and addition functions. Then the results are written to the rational number memory of the GPU.

Subtraction

Rational number subtraction of two numbers $\frac{a}{b}$ - $\frac{c}{d}$ can be written as $R = \frac{(a\times d)-(c\times b)}{b\times d}$. Thus the same as for the rational number addition applies except that instead of the addition the integer subtraction is performed.

Multiplication

Multiplication of a rational number with another is just a multiplication of the nominators and another multiplication of the denominators. Therefore the multiplication of F_1 and F_2 equals $R = \frac{a\times c}{b\times d}$.

Division

A division of two rational numbers F_1 and F_2 is the multiplication of F_1 with the multiplicative inverse of F_2. The result is R equals $\frac{a\times d}{b\times c}$. One can see that for multiplication and division the existing integer implementation can be leveraged.

4.5 Computation-to-Warp (CTW) Mapping

As it is important to understand the CTW mapping for the benchmark results in Chapter 5 this approach, used by Langer in his work, is explained here as well.

In general the approach is to perform the operations on chunks of the integers. At first one warp is assigned to one computation and the number of necessary warps to perform the complete operation is calculated. Then one warp iterates over the complete integer and performs the computation chunk-wise. This approach introduces complexity in regards to memory handling and carry propagation. However as previously [Lan15] evaluated, this divide and conquer principle scales very well for large integers and scales well on manycore architectures such as the GPU. The performance gain over traditional implementations on the CPU is significant. The fully detailed explanation of the operations can be found in Langer's paper [Lan15]. A schematic overview of this approach can be seen in 4.2

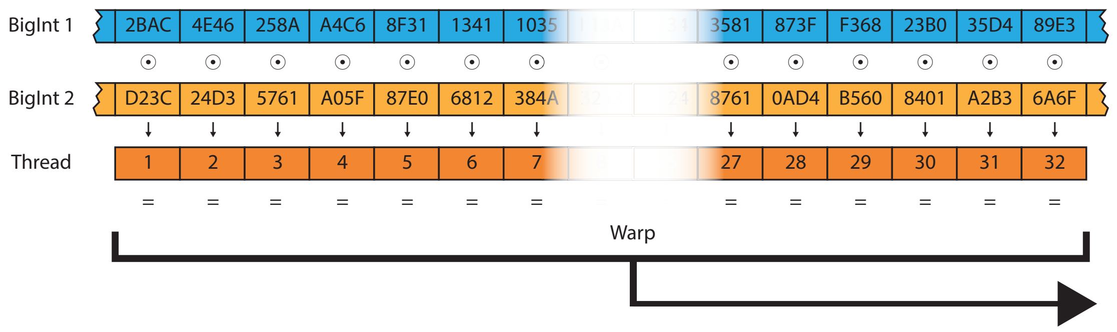

Figure 4.2: A schematic representation of CTW mapping. Showing two integers where a chunk of 32 words of each integer is processed in parallel by a full warp.

Addition

In principle the warp-wise addition takes two 32 word wide chunks of memory and each thread in the warp performs a single 32 bit integer addition. Then checks are performed to see if the operation produces an integer overflow and whether a carry needs to be propagated or generated. In the next step carry propagation for the full warp is performed and if necessary passed on to the next iteration performing computations on the next part of the integer. This procedure is repeated until the full integer addition has been performed and the position and length of the result in global memory are saved.

Subtraction

The warp-wise subtraction works exactly like the addition except that no carry for overflows needs to be detected. Instead each thread checks if the subtraction results in an integer underflow and then a carry is propagated which will be subtracted instead of added. Once again the last resulting negative carry is propagated to the next iteration.

Multiplication

The following Figure 4.3 depicts a multiplication of two numbers as one would do using the school method. This method multiplies the first factor $F_1 = 1024$ with the each of the numbers of the second factor $F_2 = 768$. For each position change in F_2 the intermediary result is shifted by one digit as well.

$$
\begin{array}{cccccc}
 & & 1 & 0 & 2 & 4 \\
 & \times & & 7 & 6 & 8 \\
\cline{2-6}
 & & 8 & 1 & 9 & 2 \\
 & 6 & 1 & 4 & 4 & \\
7 & 1 & 6 & 8 & & \\
\hline
7 & 8 & 6 & 4 & 3 & 2
\end{array}
$$

Figure 4.3: Multiplication using the school method

The idea for parallelizing this operation is to map the computation of each line to one warp. This time two carry propagation steps have to be handled. The multiplication can produce carries and the subsequent addition of all intermediary results can produce carries too which need another step to handle carry propagation.

CHAPTER 5

Results

This chapter contains a detailed explanation of the benchmark process measuring the performance of computation-to-thread parallelization compared to the warp-to-integer-chunk mapping. In addition the extension for integers of small size has been benchmarked against a state of the art library called "Library of Efficient Data types and Algorithms" or short LEDA [led15]. Furthermore the code used on the GPU has been ported to the CPU and a raw performance comparison benchmark is presented as well. The specifications for the benchmark computer can be found in Appendix A.
To keep this section short only an excerpt of data plots that show the most significant effects are shown here. In Section C in the appendix one can find the complete set of plots of all the benchmarks performed.

5.1 GPU to CPU Comparison

When comparing the GPU to the CPU the effect of costly memory transfers can be seen. Costly memory transfers means that the memory transfers to the GPU are slow. This leads to a relativization of the performance boost gained by the massive parallelization of the GPU when the computation runtimes are low. However for large runtimes such as for the multiplication the memory transfers do not affect the timings considerably. This effect can be seen when the timings for addition and subtraction are compared to the multiplication in Figure 5.2. For integers consisting of only one word the execution times of 1 millisecond and below are so low that largest part of the runtime stems from the memory transfers. In Figure 5.2 the computation times for the multiplication are increased and therefore the 28 to 38 milliseconds for memory transfer do not impact the overall performance. It can be seen that the GPU outperforms the CPU when overall execution times are high. Another scenario where the transfer to the GPU is practical is when the computation times are low but the number of computations is high enough that the memory transfers do not impact the overall timings.

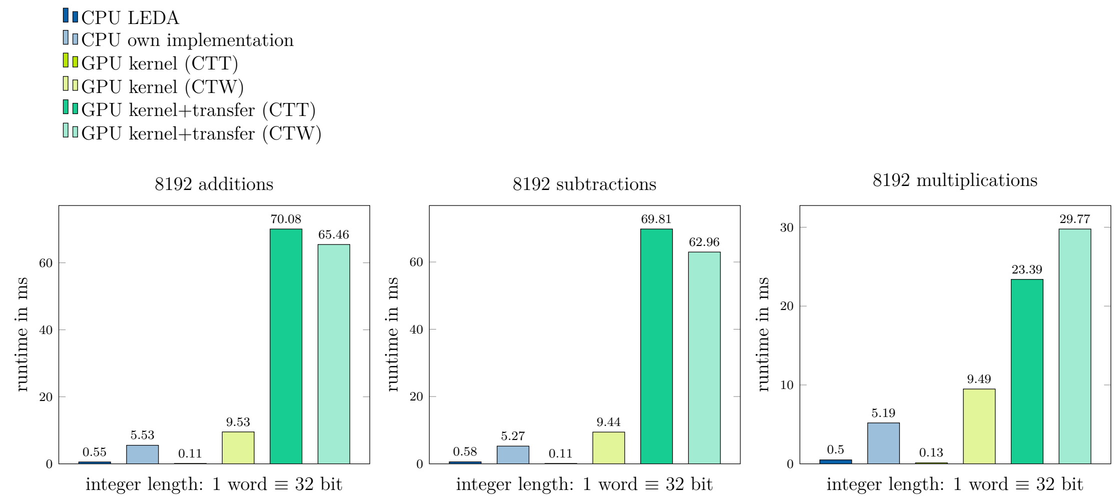

Figure 5.1: Runtime comparison of all implementations for addition, subtraction and multiplication with operands of a length of 1 word.

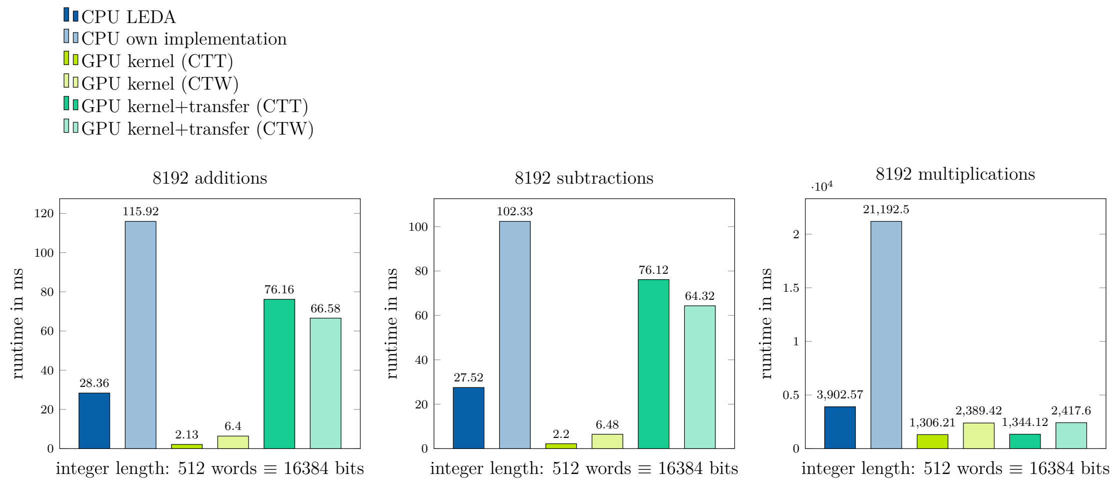

Figure 5.2: Runtime comparison of all implementations for addition, subtraction and multiplication with operands of a length of 512 words.

If one disregards memory transfers and assumes a high enough workload on the GPU that the runtime of operations exceeds the memory transfer times then the GPU outperforms the CPU. This effect is shown by Langer et al. for the CTW mapping but also applies to the novel approach to map computations to threads. Such increased runtime occurs if either the integer length is large enough or the number of computations is high. This effect can be seen righthand subplot in Figure 5.2 where the quadratic nature of the multiplication increases runtime to a level where the memory transaction can be neglected. The same effect can be seen in the rational number benchmarks Figures 5.3 and 5.4.

5.2 Rational Numbers Benchmarks

For benchmarking the runtime of rational numbers the nominator and denominator are integers that are created by an underlying normal distribution that creates random numbers around a given length and deviation. Each of these length and deviation settings is repeated 32 times and the timings are averaged.
Operations for rational numbers behave similar in runtime to simple integer operations as addition, subtraction, multiplication and division are a collection of integer base computations. This behaviour is explained in Section 4.4. The major difference is that given the circumstance that a rational number addition involves three multiplications and one addition the length of the nominator and denominator increases considerably. This stems from the fact that the worst case result length for a multiplication is the sum of the lengths of both factors. Therefore the runtime of rational number computations increases with the number of base operations. Optimizations to decrease the runtime of rational number operations are mentioned in Chapter 6. Figures 5.3 and 5.4 show the runtimes for smallest and highest amount of computations benchmarked.

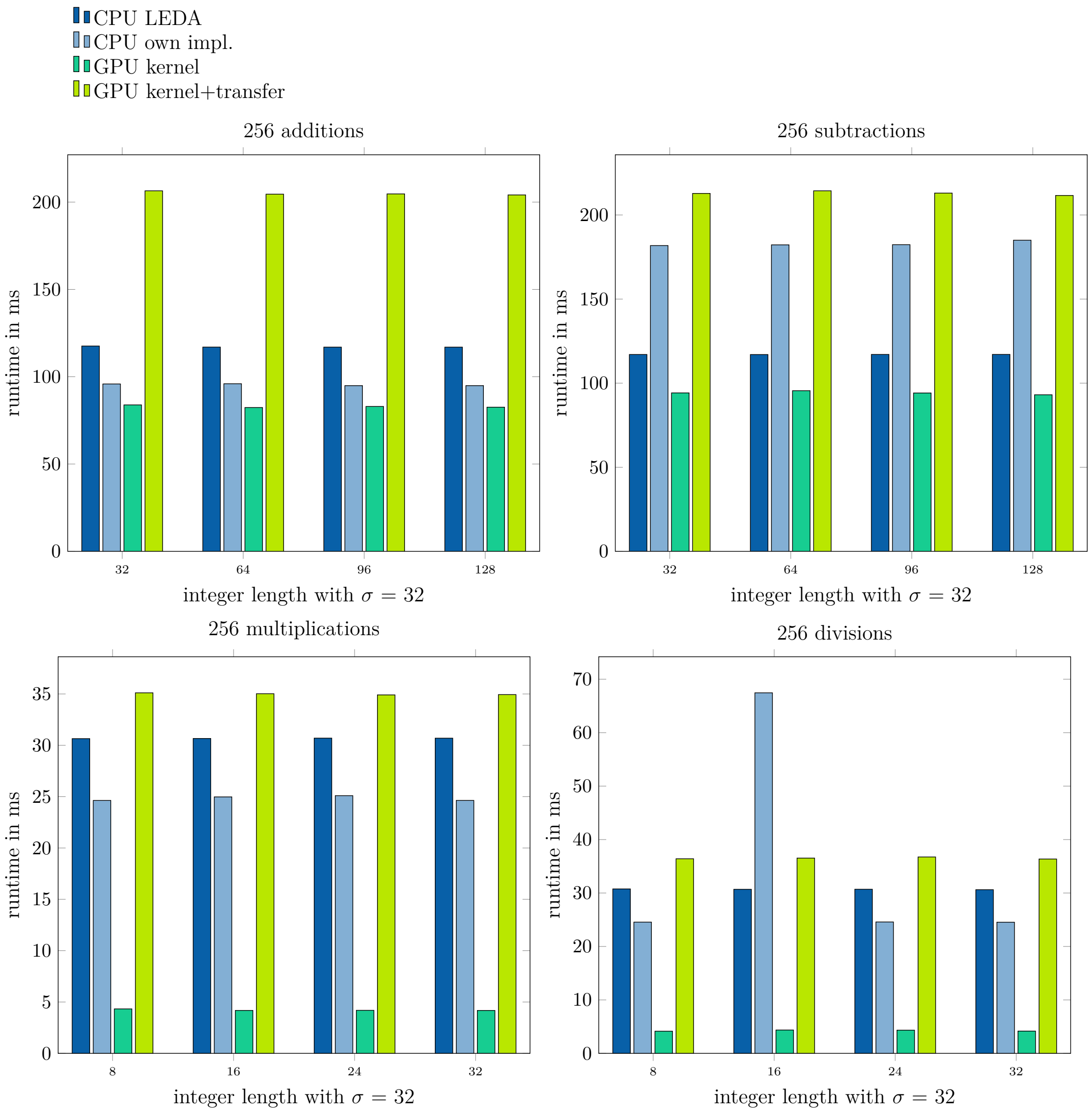

Figure 5.3: Showing the runtime performance of addition, subtraction, multiplication and division for rational numbers. The x-axis is showing the nominator and denominator length with a deviation of 32 as multiples of 32 bit. Performance timings are for 256 computations. There are not enough computations and therefore memory transfers outweigh the runtime advantage of the GPU.

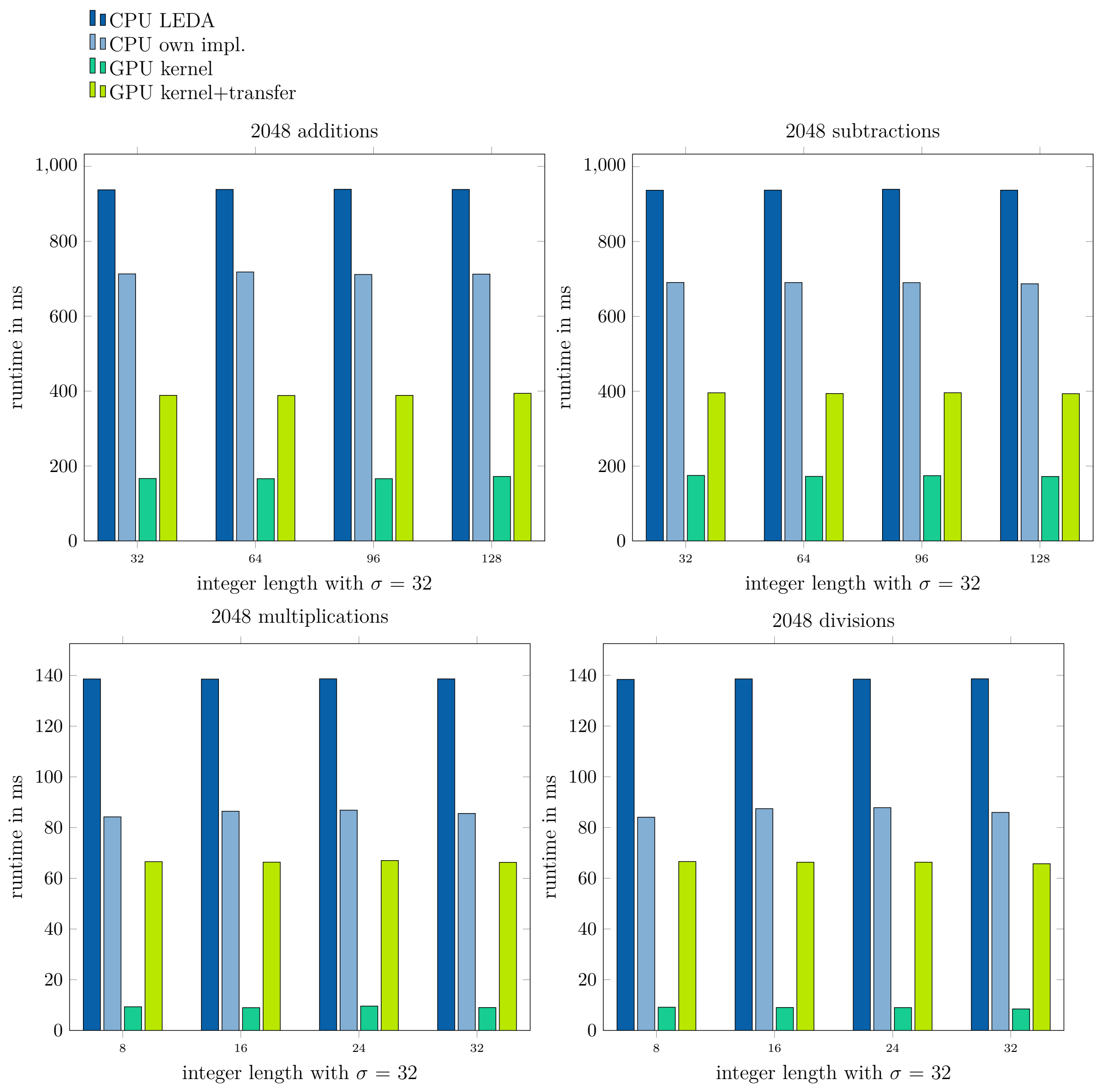

Figure 5.4: Showing the runtime performance of addition, subtraction, multiplication and division for rational numbers. The x-axis is showing the nominator and denominator length with a deviation of 32 as multiples of 32 bit. Performance timings are for 2048 computations. In this figure the memory transfers do not impact the overall performance of the GPU as the computation times are high enough to hide the transfer time.

5.3 Computation-to-warp to Computation-to-thread Comparison

To compare the performance of the CTT parallelization to the CTW approach 8192 additions, subtractions and multiplications with integers of increasing length are performed. Integers start a length of one and are increased by a multiple of two up to 512 words length. The same benchmarking behaviour of averaging 64 benchmark iterations per integer length as used to compare the GPU and CPU runtime is used for this comparison. A visible effect is the performance advantage of CTT parallelization compared to CTW parallelization for small numbers. The computation-to-thread mapping outperforms the other method throughout the full range from one word integers to 512 word long integers. However it has to be noted that the CTT mapping does not scale as well as the CTW solution and that this benchmark only considers operands of equal length. While the solution implemented by Langer does have a lower limit for the execution time of around 9 milliseconds, mapping computations to threads within a warp does not have this limit. This lower bound stems from the fact that for integers with less than 32 words not all threads within a warp are performing work and therefore computation power is lost due to inactive threads. This approach even performs better when warps are saturated as can be seen in the runtime comparison of both mapping techniques in Figure 5.5.

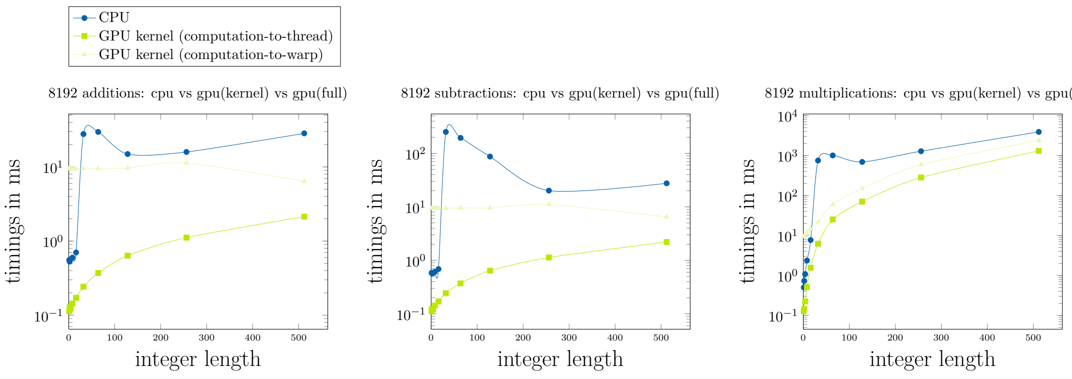

Figure 5.5: The logarithmic runtime performance for the CTW and CTT computation mappings for integers of increasing length. The integer length is in multiples of 32 bit.

5.4 Benchmark Anomalies

There is unexpected runtime behaviour of two different benchmark cases. The first case is linked to an increase in computation time for the CPU and the second are unexpected benchmark timings for the GPU multiplication.
The first effect is connected to the execution of computations on the CPU. When the word count of integers exceeds 32 the runtime of the CPU increases from less than one millisecond to 251 milliseconds in the worst case. For 64 words and more the runtime decreases and converges on the expected runtime at around 256 words. This effect can

be seen in Figure 5.6 and Figure C.2 as a peak around the length of 32 and 64 words. In order to avoid unforeseeable implementation details of the LEDA library, the GPU code is translated to run on the CPU. This behaviour occurred for the not optimized self implemented CPU code as well.

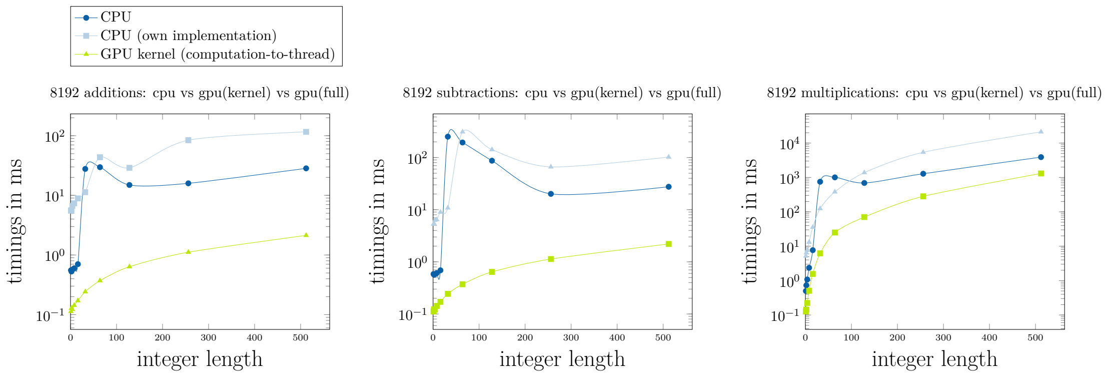

Figure 5.6: The logarithmic runtime performance of the LEDA library, the non-optimized CPU implementation and the GPU CTT mapping. Performing the computations on the GPU shows better performance than running the operations in serial on the CPU.

The second effect can be seen in Figure 5.1 and 5.2. The addition and subtraction runtime performances for the CTT mapping measuring GPU kernel execution with memory transfers is below the performance of the CTW mapping. However for the multiplication CTT outperforms CTW. This behaviour is unexpected as, for the multiplication, more memory is transferred to the GPU. The multiplication algorithm used for the CTT mapping needs memory that is set to zero. The zeroing is performed on the host side by the CPU. Compared to the CTW mapping a larger block of memory is transferred to the GPU. It is expected that transferring more memory to the GPU results in increased benchmark timings for kernel execution with memory transfers. Nonetheless the opposite for the CTT mapped multiplication occurs and runtimes are lower than for the CTW mapping.
Although there are two anomalies, the overall results show that performing computations CTT mapped outperforms the CPU and CTW mapping. Finding the source of the anomalies is beyond the scope of this work but from the Figures 5.1 and 5.2 it can be seen that these anomalies do not invalidate the overall results of this thesis.

CHAPTER 6

Limitations and Future Work

6.1 Implementing a Library

The work of this thesis is to explore a parallelization technique for integers of arbitrary length. To help others leverage the computing power of the GPU this work could be incorporated into a library. The library could include the CTW mapping technique to support good performance for integers larger than 32 words and deviating operand lengths. If both methods would be incorporated into a library it would provide others with the ability to run high numbers of computations of integers and rational numbers of arbitrary precision. The implementation could even hide the fact that two different parallelization techniques are used and could choose the appropriate method automatically.

6.2 Greatest Common Divisor for Rational Numbers

As stated in Section 4.4 the results of the arithmetic operations grow with each operation. This behaviour results in very high memory usage and longer memory transfers. To reduce this overhead and be able to perform more computations it is necessary to implement the greatest common divisor to reduce numbers before computation.

6.3 Reduced Memory Transfers

The memory transfer methods used to benchmark the technique are not fully optimized and can be improved. Memory transfers as implemented to benchmark the CTT mapping transferred the full memory block to and from the GPU. By only transferring the operands to the GPU and the resulting numbers back from it, memory transfer timings could be reduced. One can see in Section 5 that, especially for small numbers, the memory transfer significantly impacts the overall GPU performance. By optimizing this step necessary for executing on the GPU one can improve the runtime for small numbers considerably.

6.4 Memory Layout Middleware

A step towards general usability of the library could be to introduce memory constructs that are more intuitive for the programmer. Instead of using *structs* holding pointers higher level representations of integers, rational numbers and computations could be introduced. To keep the current memory representations for the GPU a layer that translates between high level representations and memory look-up optimized *structs* needs to be implemented. This layer would increase the ease of use of the library while keeping the existing codebase functional and performing optimal.

6.5 Functionality Expansion

To come one step closer to a full-fledged library the existing implementation should be extended by supporting more algorithms and number formats. This can include floats with extended precision [LHL10] or real number types as well as algorithms to support geometric calculations.

6.6 Anomaly Investigation

As explained in Chapter 5 there are two anomalies linked to the runtime performance of the implementation. The reason for these anomalies needs to be detected and the implementation improved to avoid unexpected behaviour. Benchmarks need to be run again to gather data that is not influenced by unexpected behaviour and shows the runtime improvement more clearly.

CHAPTER 7

Conclusion

This thesis explored a novel mapping technique of arbitrary precision integer computations to threads of a GPU. Additionally benchmarks to compare the novel approach to a state of the art implementation on the CPU using the LEDA library [led15] and to Langer's [Lan15] approach are performed. This benchmark which runs computations for integers of increasing length from 32 to 16384 bit shows that the approach presented in this thesis is indeed a viable parallelization method not only for integers with less than 32 words but also if the operands of integer computations are equal in size.
Additionally a first step towards an implementation for general use for computations on the GPU has been made by implementing the rational number format. This rational number format was also benchmarked and compared to the implementation of the LEDA library. It is shown that the GPU outperforms the CPU not only for integers of arbitrary length but also for rational numbers if the number of available processors can be saturated and the computations themselves are expensive in time.

APPENDIX A

Test System Specifications

- Operating System: Windows 7 64 bit
- CPU: Intel 4770k
  - Number of Cores: 4
  - Number of Threads: 8
  - Base Frequency: 3.5 Ghz
  - Cache: 8 MB
  - Memory Bandwidth: 25.6 GB/s
- RAM: 16 GB
- GPU Specifications: Nvidia Titan X
  - Manufacturer: EVGA
  - CUDA Cores: 3072 CUDA Cores
  - Base Clock: 1127 MHz
  - Video RAM: 12288 MB, 384 bit GDDR5
  - Memory Bandwidth: 336.5 GB/s

APPENDIX B

Algorithm Pseudo Code

Algorithm B.1: Arbitrary Precision Integer Multiplication Loop

Input: The position of the first operand p_1 and the second operand p_2. The length of both operands s_1, s_2. p_r as the offset for the result in global memory. A function `GetWord()` to retrieve a word from global memory. Another function `SetWord()` to save a word to global memory. Two functions `GetLowerHalf()` and `GetUpperHalf()` to get the lower or upper 32 bit of a 64 bit integer.

Output: The updated global memory.

```
for $i \leftarrow 0; i < s_1; i \leftarrow i + 1$ do
    $j \leftarrow 0$;                                  // Set inner loop variable
    $c \leftarrow 0$;                                  // Initialize carry
    $w_1 \leftarrow$ GetWord($p_1 + i$);               // Set first operand word
    while $j < s_2$ or $c \neq 0$ do
        $w_2 \leftarrow 0$;                            // Init second operand word
        if $j < s_2$ then
            $w_2 \leftarrow$ GetWord($p_2 + j$);       // Set second operand word
        end
        $t \leftarrow$ GetWord($p_r + j + i$);         // Get previous temp result
        $t \leftarrow w_1 \times w_2 + t + c$;         // Calculate 64 bit result
        $l \leftarrow$ GetLowerHalf($t$);              // Get lower 32 bit of result
        SetWord($p_r + j + i, l$);                     // Set result in memory
        $c \leftarrow$ GetUpperHalf($t$);              // Set carry
        $j \leftarrow j + 1$;                          // Increase inner loop counter
    end
end
```

APPENDIX C

Benchmark Figures

C.1 Integer Benchmark Plots

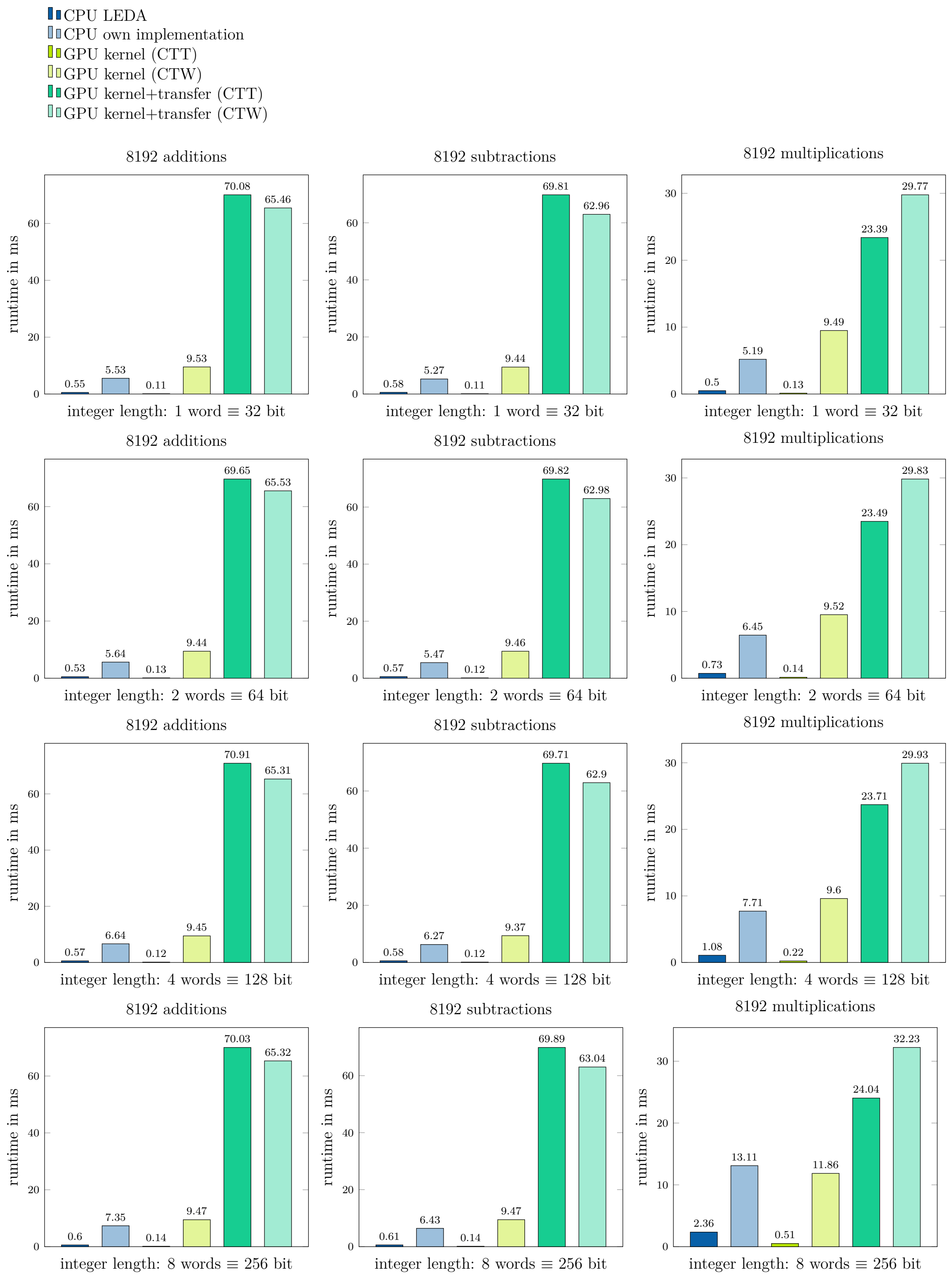

Figure C.1: Plots for the addition, subtraction and multiplication of integers of a length of 1 to 8 words. The bars show the CPU and GPU timings for all measured implementations

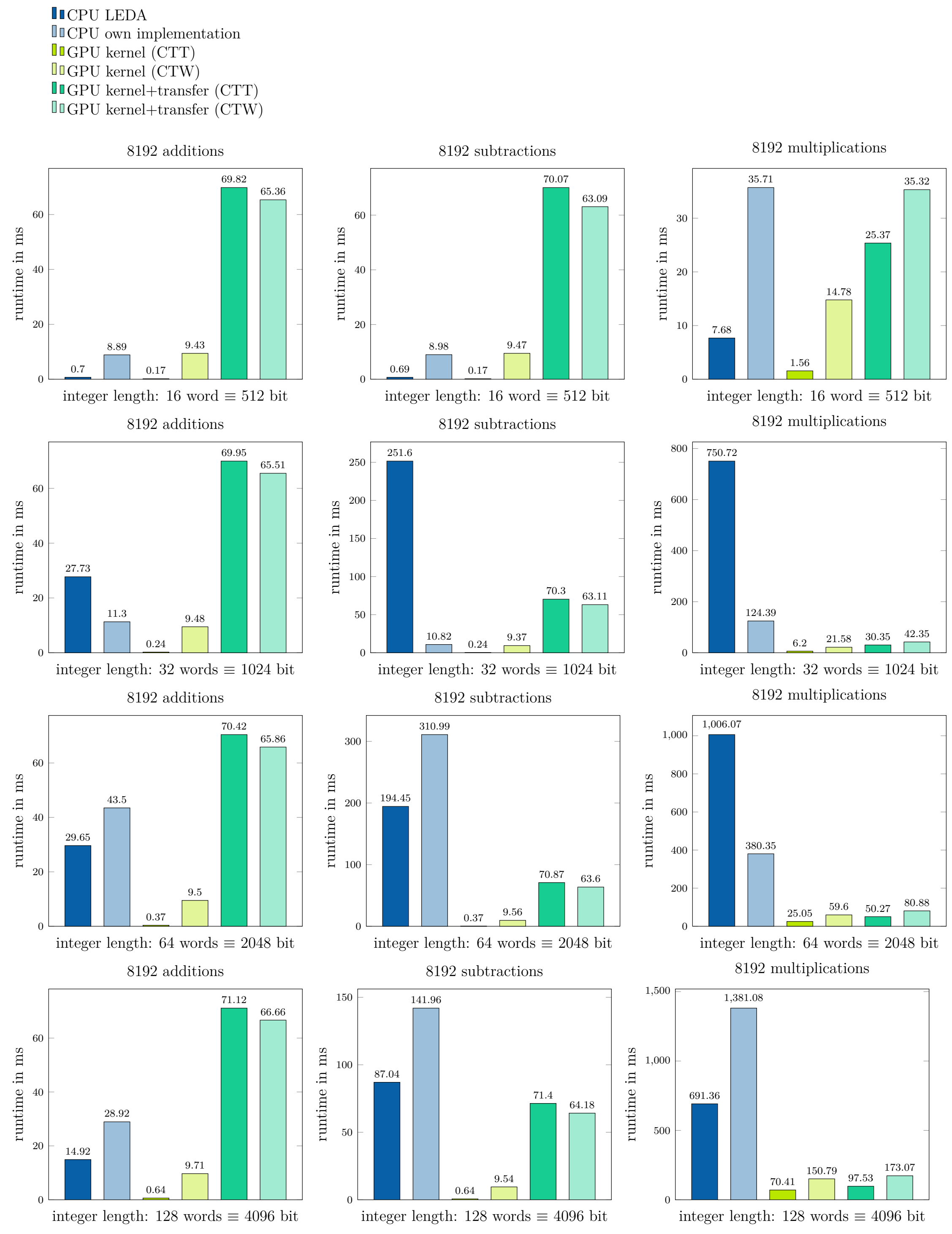

Figure C.2: Plots for the addition, subtraction and multiplication of integers of a length of 16 to 128 words. The bars show the CPU and GPU timings for all measured implementations.

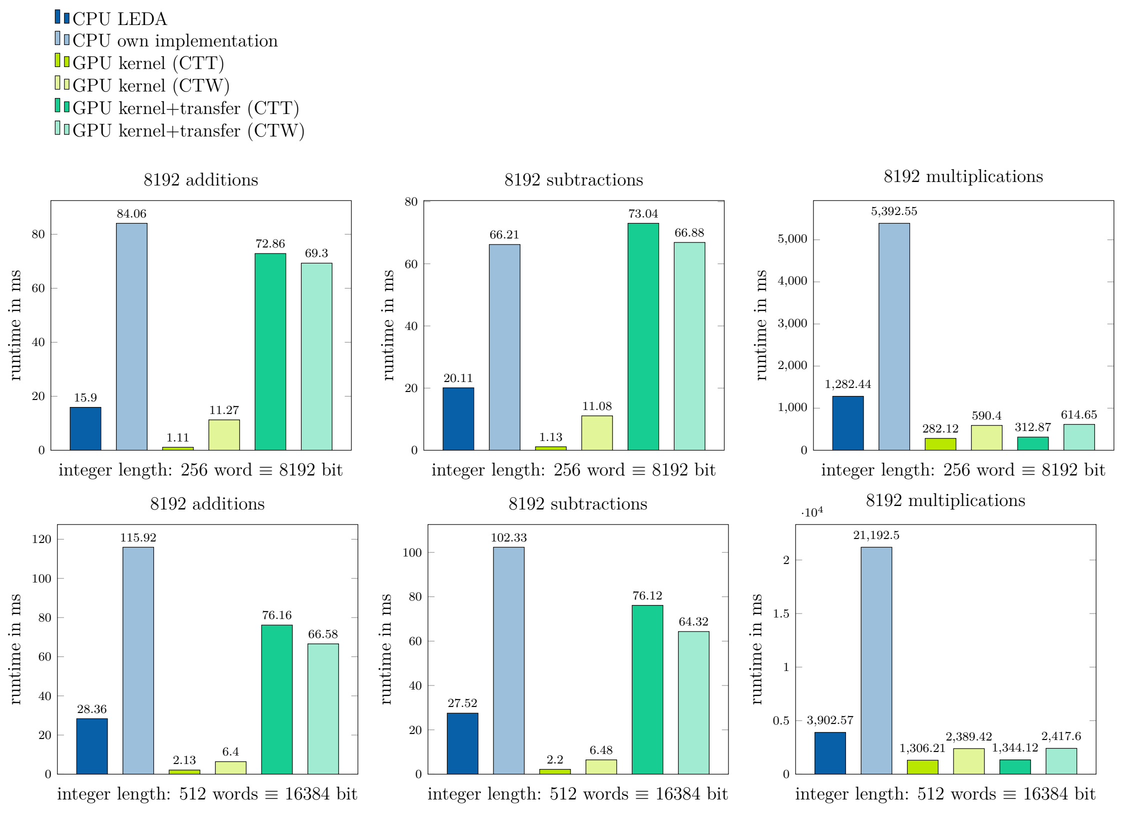

Figure C.3: Plots for the addition, subtraction and multiplication of integers of a length of 256 to 512 words. The bars show the CPU and GPU timings for all measured implementations.

C.2 Rational Number Plots

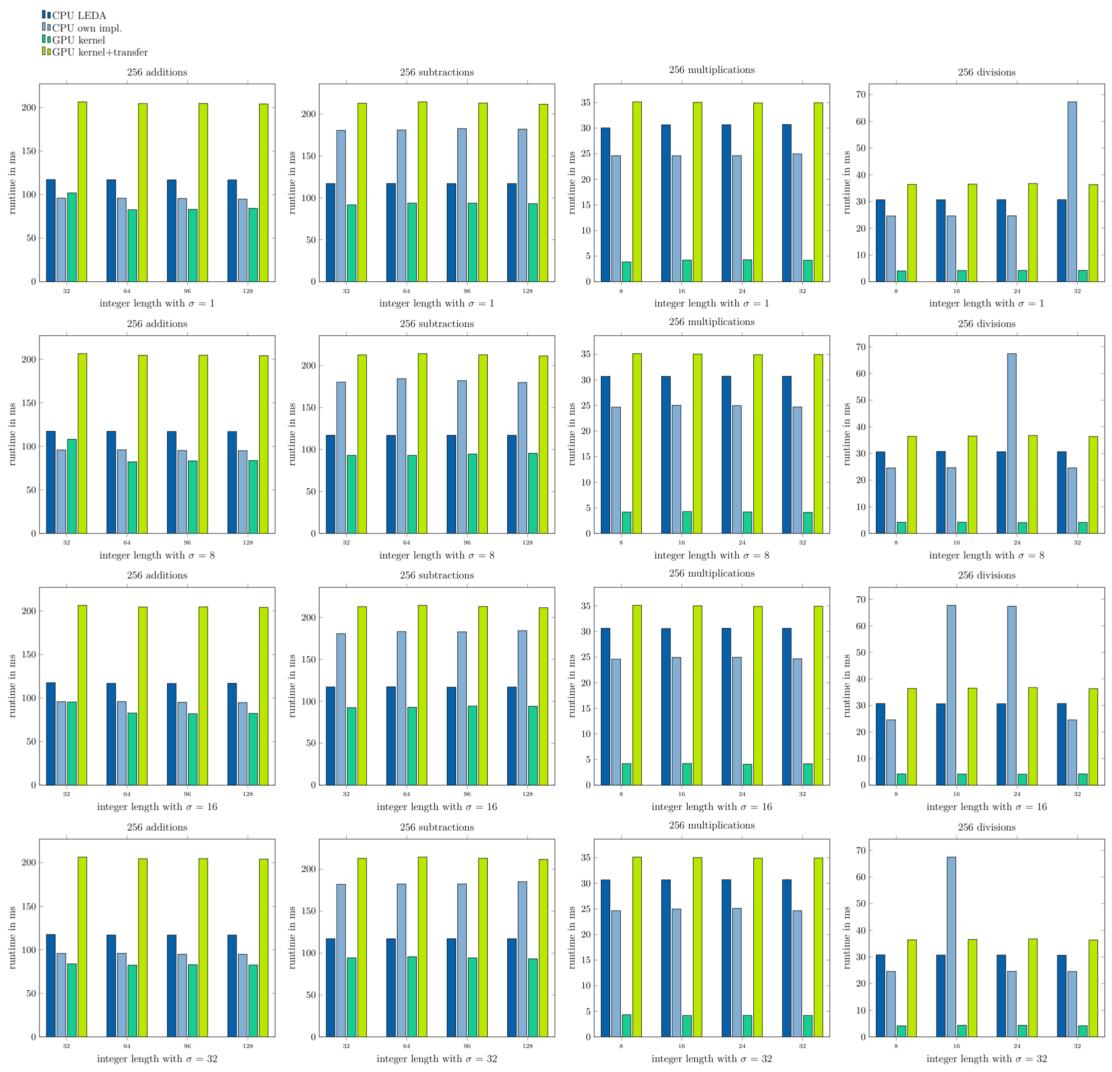

Figure C.4: Benchmark timings for 256 rational number computations addition, subtraction, multiplication and division. On the x-axis are the lengths of the nominator and denominator in multiples of 32 bit. The deviation of the length of the nominator and denominator are in each row from $\sigma = 1$ to $sigma = 32$.

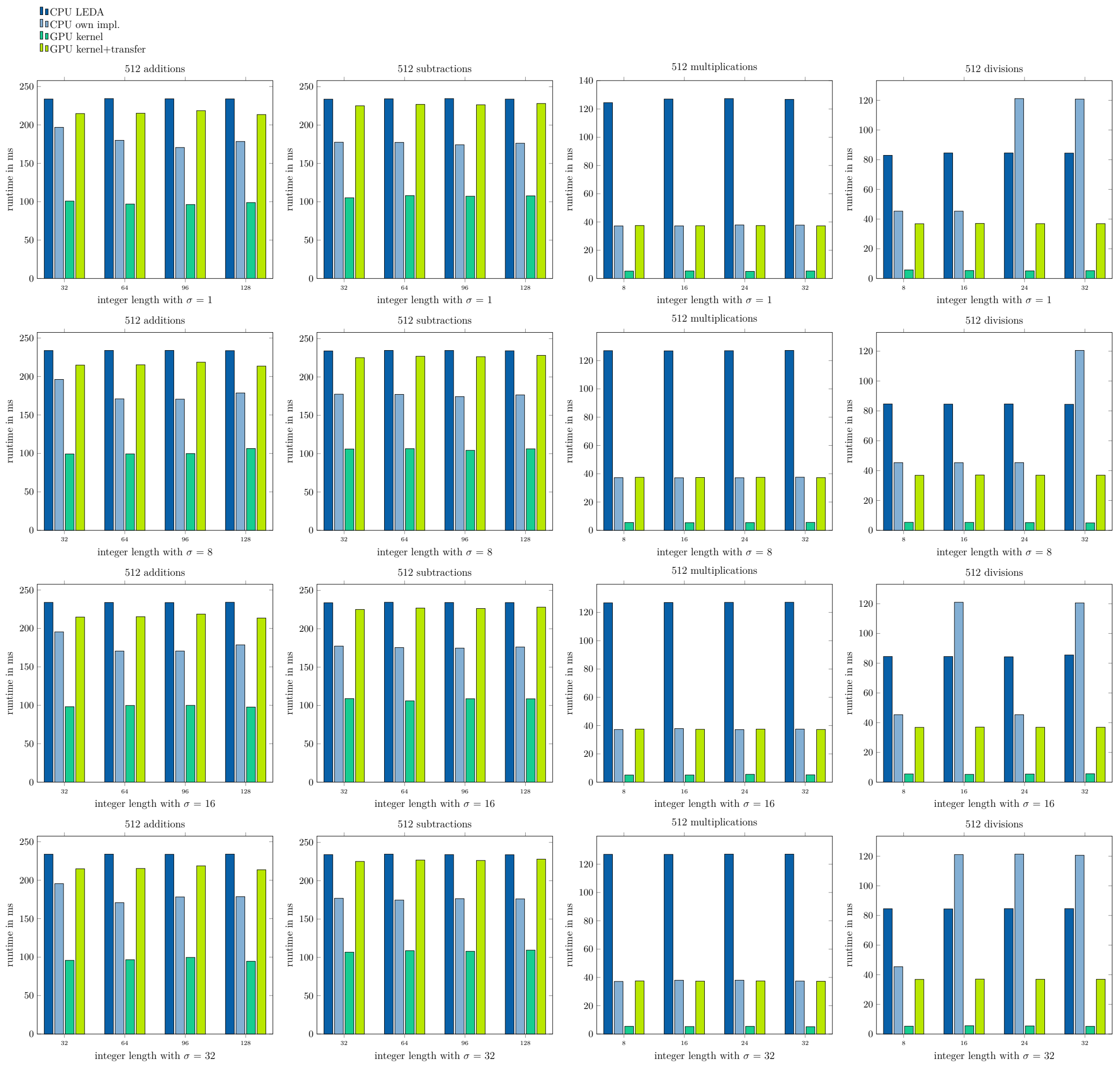

Figure C.5: Benchmark timings for 512 rational number computations addition, subtraction, multiplication and division. On the x-axis are the lengths of the nominator and denominator in multiples of 32 bit. The deviation of the length of the nominator and denominator are in each row from $\sigma = 1$ to $sigma = 32$.

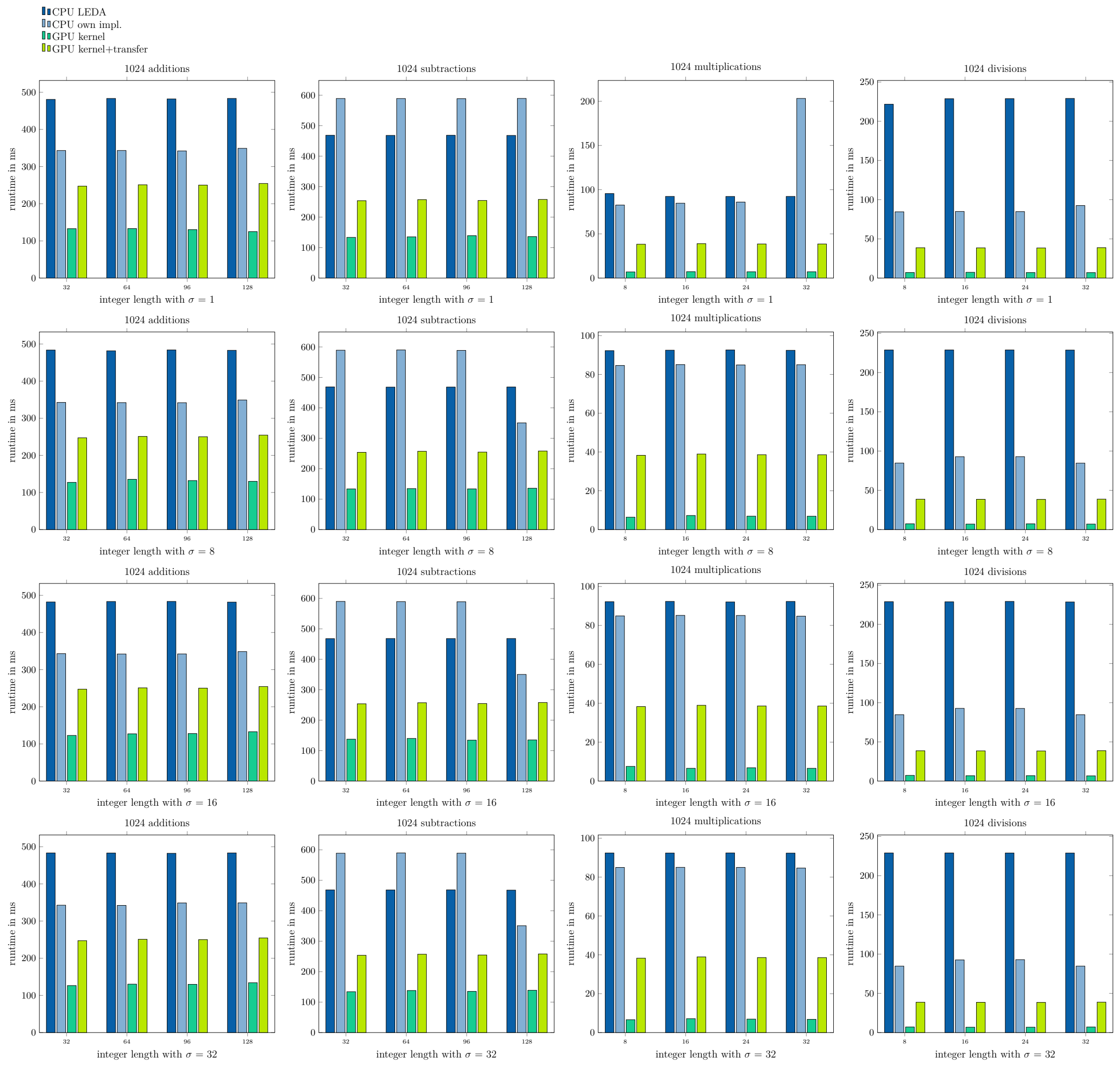

Figure C.6: Benchmark timings for 1024 rational number computations addition, subtraction, multiplication and division. On the x-axis are the lengths of the nominator and denominator in multiples of 32 bit. The deviation of the length of the nominator and denominator are in each row from $\sigma = 1$ to $sigma = 32$.

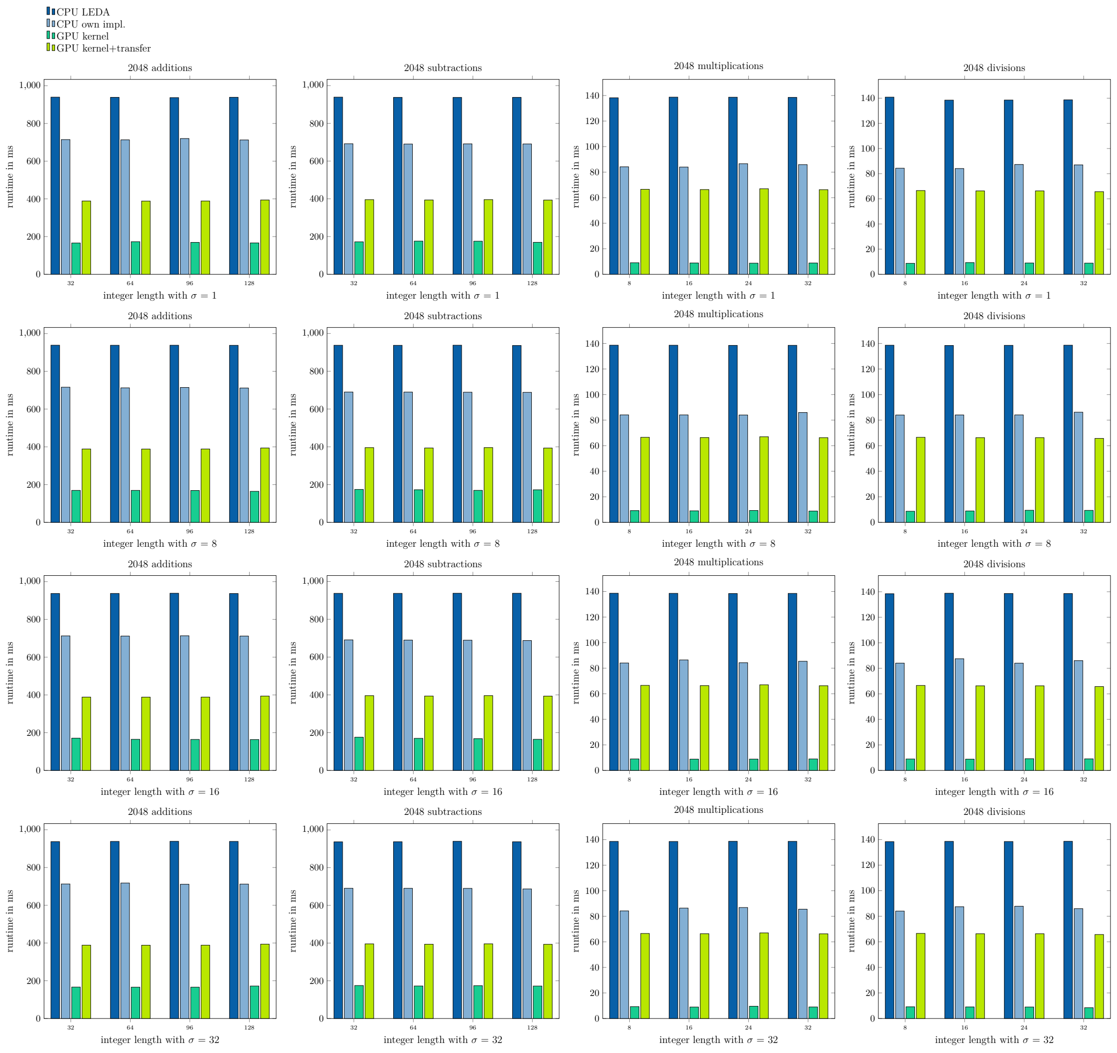

Figure C.7: Benchmark timings for 2048 rational number computations addition, subtraction, multiplication and division. On the x-axis are the lengths of the nominator and denominator in multiples of 32 bit. The deviation of the length of the nominator and denominator are in each row from $\sigma = 1$ to $sigma = 32$.

List of Figures

List of Tables

List of Algorithms

Acronyms

CPU Central processing unit. v, vii, 1–3, 7, 23, 24, 28, 29, 33, 49

CTT Computation-to-Thread. vii, 16, 17, 24, 28, 29, 31, 41–43, 49

CTW Computation-to-Warp. vii, 20, 21, 24, 28, 29, 31, 41–43, 49

CUDA Compute Unified Device Architecture. vii, 3–5, 7, 8, 15, 49

DSP digital signal processor. 3

FPGA field programmable gate array. 3

GPGPU General-Purpose Computing on Graphics Processing Units. vii, 2, 3, 5, 7

GPU Graphics processing unit. v, vii, 1–3, 7–10, 15, 16, 19, 20, 23, 24, 26–29, 31–33, 49

LEDA Library of Efficient Data types and Algorithms. 23

MIMD multiple instruction multiple data. 2

SIMD single instruction multiple data. 2, 7

SM streaming multi-processors. 3, 4

Bibliography

[amd15] AMD OpenCL Zone - Tools and SDKs, 2015. `http://developer.amd.com/tools-and-sdks/opencl-zone/`.

[CMS12] D. Ciresan, U. Meier, and J. Schmidhuber. Multi-column deep neural networks for image classification. In *Computer Vision and Pattern Recognition (CVPR), 2012 IEEE Conference on*, pages 3642–3649, June 2012.

[DYDA12] G.E. Dahl, Dong Yu, Li Deng, and A. Acero. Context-dependent pre-trained deep neural networks for large-vocabulary speech recognition. *Audio, Speech, and Language Processing, IEEE Transactions on*, 20(1):30–42, Jan 2012.

[Gee05] David Geer. Chip makers turn to multicore processors. *Computer*, 38(5):11–13, May 2005.

[HJPM10] Sangjin Han, Keon Jang, KyoungSoo Park, and Sue Moon. Packetshader: a gpu-accelerated software router. *SIGCOMM Comput. Commun. Rev.*, 41(4):–, August 2010.

[JSD$^+$14] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional architecture for fast feature embedding. In *Proceedings of the ACM International Conference on Multimedia*, MM '14, pages 675–678, New York, NY, USA, 2014. ACM.

[Khr15a] Khronos Group. About The Khronos Group, 2015. `https://www.khronos.org/about/`.

[Khr15b] Khronos Group. The open standard for parallel programming of heterogeneous systems, 2015. `https://www.khronos.org/opencl/`.

[L$^+$04] David Luebke et al. General-purpose computation on graphics hardware. In *Workshop, SIGGRAPH*, 2004.

[Lan15] Bernhard Langer. Arbitrary-precision arithmetics on the gpu. In Michael Wimmer, Jiri Hladuvka, and Martin Ilcik, editors, *19th Central European Seminar on Computer Graphics*, CESCG '15, pages 25–31. Vienna University of Technology, apr 2015. Supervised by Thomas Auzinger. Non-peer reviewed.

[led15] Library of Efficient Data types and Algorithms, 2015. `http://algorithmic-solutions.com/`.

[LHL10] Mian Lu, Bingsheng He, and Qiong Luo. Supporting extended precision on graphics processors. In *Proceedings of the Sixth International Workshop on Data Management on New Hardware*, DaMoN '10, pages 19–26, New York, NY, USA, 2010. ACM.

[Moo98] G.E. Moore. Cramming more components onto integrated circuits. *Proceedings of the IEEE*, 86(1):82–85, Jan 1998.

[nvi15a] Nvidia CUDA Zone - cuBLAS Library, 2015. `https://developer.nvidia.com/cublas`.

[nvi15b] Nvidia CUDA Zone - cuFFT Library, 2015. `https://developer.nvidia.com/cufft`.

[nvi15c] Nvidia CUDA Zone - GPU-Accelerated Libraries, 2015. `https://developer.nvidia.com/gpu-accelerated-libraries`.

[nvi15d] Nvidia CUDA Zone - Thrust Library, 2015. `https://developer.nvidia.com/thrust`.

[Nvi15e] CUDA Nvidia. Nvidia cuda c programming guide. *NVIDIA Corporation*, pages 115–118, 2015.

[OHL$^+$08] J.D. Owens, M. Houston, D. Luebke, S. Green, J.E. Stone, and J.C. Phillips. Gpu computing. *Proceedings of the IEEE*, 96(5):879–899, May 2008.

[PJ14] Valentina Popescu and Mioara Joldes. Towards fast and certified multiple-precision libraries. 2014.

[pyr15] Pyrit GPU Cracker Project Page, 2015. `https://code.google.com/p/pyrit/`.

[SA 15] SA Transcripts. Intel's (INTC) CEO Brian Krzanich on Q2 2015 Results - Earnings Call Transcript, 2015. `http://seekingalpha.com/article/3329035-intels-intc-ceo-brian-krzanich-on-q2-2015-results-earnings-call-transcript?page=2`.

[TDF$^+$02] J.M. Tendler, J.S. Dodson, J.S. Fields, H. Le, and B. Sinharoy. Power4 system microarchitecture. *IBM Journal of Research and Development*, 46(1):5–25, Jan 2002.

[VAP$^+$08] Giorgos Vasiliadis, Spiros Antonatos, Michalis Polychronakis, Evangelos P Markatos, and Sotiris Ioannidis. Gnort: High performance network intrusion detection using graphics processors. In *Recent Advances in Intrusion Detection*, pages 116–134. Springer, 2008.

[WACS11] Changchang Wu, S. Agarwal, B. Curless, and S.M. Seitz. Multicore bundle adjustment. In *Computer Vision and Pattern Recognition (CVPR), 2011 IEEE Conference on*, pages 3057–3064, June 2011.