12-20-2016 12:00 AM

# Fast Fourier Transforms over Prime Fields of Large Characteristic and their Implementation on Graphics Processing Units

Davood Mohajerani, *The University of Western Ontario*

Supervisor: Dr. Marc Moreno Maza, *The University of Western Ontario*

A thesis submitted in partial fulfillment of the requirements for the Master of Science degree in Computer Science

# Abstract

Prime field arithmetic plays a central role in computer algebra and supports computation in Galois fields which are essential to coding theory and cryptography algorithms. The prime fields that are used in computer algebra systems, in particular in the implementation of modular methods, are often of small characteristic, that is, based on prime numbers that fit on a machine word. Increasing precision beyond the machine word size can be done via the Chinese Remainder Theorem or Hensel's Lemma.

In this thesis, we consider prime fields of large characteristic, typically fitting on $n$ machine words, where $n$ is a power of 2. When the characteristic of these fields is restricted to a subclass of the generalized Fermat numbers, we show that arithmetic operations in such fields offer attractive performance both in terms of algebraic complexity and parallelism. In particular, these operations can be vectorized, leading to efficient implementation of fast Fourier transforms on graphics processing units.

**Keywords:** Fast Fourier transforms, finite fields of large characteristic, graphics processing units

Acknowlegements

First and foremost, I would like to offer my sincerest gratitude to my supervisor Professor Marc Moreno Maza, I am very thankful for his great advice and support.

It is my honor to have Professor John Barron, Professor Dan Christensen, and Professor Mark Daley as the examiners. I am grateful for their insightful comments and questions.

I would like to thank the members of Ontario Research Center for Computer Algebra and the Computer Science Department of the University of Western Ontario. Specially, I am thankful to my colleagues Dr. Ning Xie, Dr. Masoud Ataei, and Egor Chesakov for proofreading chapters of my thesis.

Finally, I am very thankful to my family and friends for their endless support.

# Contents

# List of Algorithms

# List of Figures

# List of Tables

# Chapter 1

# Introduction

Prime field arithmetic plays a central role in computer algebra and supports computation in Galois fields which are essential to coding theory and cryptography algorithms. In computer algebra, the so-called *modular methods* are the main application of prime field arithmetic. Let us give a simple example of such methods.

Consider a square matrix $A$ of order $n$ with coefficients in the ring $\mathbb{Z}$ of integers. It is well-known that $\det(A)$, the determinant of $A$, can be computed in at most $2n^3$ arithmetic operations in the field $\mathbb{Q}$ of rational numbers, by means of Gaussian elimination. However the cost of each of those operations is not the same and, in fact, depends on the bit size of the rational numbers involved. It can be proved that, if $B$ is the maximum absolute value of a coefficient in $A$ then computing the determinant of $A$ directly (that is, over $\mathbb{Z}$) can be done within $\mathcal{O}(n^5\,(\log n + \log B)^2)$ machine-word operations, see the landmark book [24]. If a modular method is used, based on the Chinese Remainder Theorem (CRT), one can reduce the cost to $\mathcal{O}(n^4 \log^2(nB)\,(\log^2 n + \log^2 B))$ machine-word operations.

Let us explain how this works. Let $d$ be the determinant of $A$ and let us choose a prime number $p \in \mathbb{Z}$ such that the absolute value $\mid d \mid$ of $d$ satisfies

$$2 \mid d \mid < p.$$

Let $r$ be the determinant of $A$ regarded as a matrix over $\mathbb{Z}/p\mathbb{Z}$ and let us represent the elements of $\mathbb{Z}/p\mathbb{Z}$ within the symmetric range $[-\frac{p-1}{2} \cdots \frac{p-1}{2}]$. Hence we have

$$-\frac{p}{2} < r < \frac{p}{2} \quad \text{and} \quad -\frac{p}{2} < d < \frac{p}{2} \tag{1.1}$$

leading to

$$-p < d - r < p \tag{1.2}$$

Observe that $\det(A)$ is a polynomial expression in the coefficients of $A$. For instance with $n = 2$ we have

$$\det(A) \ = \ a_{11}\,a_{22} - a_{12}\,a_{21}. \tag{1.3}$$

Denoting by $\overline{x}^p$ the residue class in $\mathbb{Z}/p\mathbb{Z}$ of any $x \in \mathbb{Z}$, we have

$$\overline{x+y}^p \ = \ \overline{x}^p + \overline{y}^p \quad \text{and} \quad \overline{xy}^p \ = \ \overline{x}^p\overline{y}^p, \tag{1.4}$$

for all $x, y \in \mathbb{Z}$. It follows for $n = 2$, and using standard notations, that we have

$$\overline{\det(A)}^p \ = \ \overline{a_{11}}^p \ \overline{a_{22}}^p - \overline{a_{12}}^p \ \overline{a_{21}}^p. \tag{1.5}$$

More generally, we have

$$\overline{\det(A)}^p \ = \ \det(A \mod p), \tag{1.6}$$

that is, $d \equiv r \mod p$. This with Relation (1.2) leads to

$$d = r. \tag{1.7}$$

In summary, the determinant of $A$ as a matrix over $\mathbb{Z}$ is equal to the determinant of $A$ regarded as a matrix over $\mathbb{Z}/p\mathbb{Z}$ provided that $2 \mid d \mid < p$ holds. Therefore, the computation of the determinant of $A$ as a matrix over $\mathbb{Z}$ can be done modulo $p$, which provides a way of controlling expression swell in the intermediate computations. See the introduction of Chapter 5 in [23] for a discussion of this phenomenon of expression swell in the intermediate computations.

But if $d$ is what we want to compute, the condition $2 \mid d \mid < p$ is not that helpful for choosing $p$. However, *Hadamard's inequality* tells us that, if $B$ is the maximum absolute value of an entry of $A$, then we have

$$\mid d \mid \leq \ n^{n/2}\,B^n. \tag{1.8}$$

One can then choose a prime number $p$ satisfying $2n^{n/2}\,B^n < p$. Of course, such prime may be very large and thus the expected benefit of controlling expression swell may be limited.

An alternative approach is to consider pairwise different prime numbers $p_1, \ldots, p_e$ such that their product exceeds $2n^{n/2}\,B^n$, and each of them fits on a machine-word. Then, computing the determinants of $A$ regarded as a matrix over $\mathbb{Z}/p_1\mathbb{Z}, \ldots, \mathbb{Z}/p_e\mathbb{Z}$ leads to values $r_1, \ldots, r_e$, respectively. Finally, applying the CRT yields $d$.

The advantage of this alternative approach is that for a prime number $p$ fitting on the machine-word of computer, arithmetic operations modulo $p$ can be implemented efficiently using hardware integer operations.

However, using machine-word size, thus small, prime numbers has also serious inconveniences in certain modular methods, in particular for solving systems of non-linear equations. Indeed, in such circumstances, the so-called *unlucky primes* are to be avoided, see for instance [1, 9].

For an example of a modular method incurring unlucky primes, let us consider the simple problem of computing a Greatest Common Divisor (GCD) of two univariate polynomials with integer coefficients. Let $f = f_n x^n + \cdots + f_0$ and $g = g_m x^m + \cdots + g_0$ be polynomials in $x$, with respective degrees $n$ and $m$, and with coefficients in a unique factorization domain (UFD) $R$. The following matrix is called the *Sylvester matrix* of $f$ and $g$.

$$\mathrm{Sylv}(f,g) \;=\; \left(\begin{array}{ccccccccc}
f_0 & & & & & g_0 & & & \\
\vdots & \ddots & & & & \vdots & & & \\
f_{n-i} & & f_0 & & & \vdots & \ddots & & \\
\vdots & \ddots & \vdots & \ddots & & g_{m-i} & \ddots & \ddots & g_0 \\
f_n & & f_{n-i} & & f_0 & \vdots & & \ddots & \vdots \\
 & \ddots & \vdots & \ddots & \vdots & g_m & \ddots & & \vdots \\
 & & f_n & & f_{n-i} & & & \ddots & g_{m-i} \\
 & & & \ddots & \vdots & & & & \vdots \\
 & & & & f_n & & & & g_m
\end{array}\right) \tag{1.9}$$

Its determinant is an element of $R$ called the *resultant* of $f$ and $g$. This determinant is usually denoted by $\mathrm{res}(f,g)$ and enjoys the following property: a GCD $h$ of $f$ and $g$ has degree zero (that is, $h$ is simply an element of $R$) if and only the $\mathrm{res}(f,g) \neq 0$ holds. In other words, $f$ and $g$ have a non-trivial GCD (that is, a GCD of positive degree) if and only the $\mathrm{res}(f,g) = 0$ holds.

Assume now that $R$ is the ring $\mathbb{Z}$ of the integer numbers and that $\mathrm{res}(f,g) \neq 0$ holds. Suppose that this latter fact is not known and that one is computing a GCD of $f$ and $g$ by means of a modular method based on the CRT. More precisely, we are computing GCDs of $f$ and $g$ modulo sufficiently many prime numbers $p_1, \ldots, p_e$, obtaining polynomials

$h_1, \ldots, h_e$ in $\mathbb{Z}/p_1\mathbb{Z}[x], \ldots, \mathbb{Z}/p_e\mathbb{Z}[x]$. If none of the prime numbers $p_1, \ldots, p_e$ divides $\mathrm{res}(f/h, g/h)$, nor the leading coefficients of $f_n$ and $g_m$, then combining $h_1, \ldots, h_e$ by CRT yields a GCD of $f$ and $g$ (which, under the assumption $\mathrm{res}(f, g) \neq 0$ turns out to be a constant). However, if one of the prime numbers $p_1, \ldots, p_e$, say $p_i$, divides $\mathrm{res}(f/h, g/h)$ (even if it does not divide $f_n$ nor $g_m$) then $h_i$ has a positive degree. It follows that $h_i$ is not a modular image of a GCD of $f$ and $g$ in $\mathbb{Z}[x]$. Therefore, this prime $p_i$ should not be used in our CRT scheme and for this reason is called *unlucky*.

Note that as the coefficients of $f$ and $g$ grow, so will $\mathrm{res}(f, g)$. As a consequence, small primes are likely to be unlucky for input data with large coefficients. While there are tricks to overcome the *noise* introduced by unlucky primes, this become a serious computational bottleneck, as raised in [2], in an application of polynomial system solving to Hilbert's 16-th Problem. To summarize, certain modular methods, when applied to challenging problems, require the use of prime numbers that do not necessarily fit on a machine-word. This observation motivates the work presented in this thesis.

In this thesis, we consider prime fields of large characteristic, typically fitting on $k$ machine words, where $k$ is a power of 2. For those modular methods in polynomial system solving that require such big prime numbers, one of the most fundamental operations is the Discrete Fourier transform (DFT) of a polynomial. Here again, we refer to the book [24].

Consider a prime field $\mathbb{Z}/p\mathbb{Z}$ and $N$, a power of 2, dividing $p - 1$. Then, the finite field $\mathbb{Z}/p\mathbb{Z}$ admits a $N$-th primitive root of unity; let us denote by $\omega$ such an element of $\mathbb{Z}/p\mathbb{Z}$. Let $f \in \mathbb{Z}/p\mathbb{Z}[x]$ be a polynomial of degree at most $N - 1$. Then, computing the DFT of $f$ at $\omega$ produces the values of $f$ at the successively powers of $\omega$, that is, $f(\omega^0), f(\omega^1), \ldots f(\omega^{N-1})$. Using an asymptotically fast algorithm, namely a fast Fourier transform (FFT), this calculation amounts to:

1. $N \log(N)$ additions in $\mathbb{Z}/p\mathbb{Z}$,
2. $(N/2) \log(N)$ multiplications by a power of $\omega$ in $\mathbb{Z}/p\mathbb{Z}$.

If the bit-size of $p$ is $k$ machine words, then

1. each addition in $\mathbb{Z}/p\mathbb{Z}$ costs $O(k)$ machine-word operations,
2. each multiplication by a power of $\omega$ costs $O(k^2)$ machine-word operations.

Therefore, multiplication by a power of $\omega$ becomes a bottleneck as $k$ grows.

To overcome this difficulty, we consider the following trick proposed by Martin Fürer in [12, 13]. We assume that $N = K^e$ holds for some "small" $K$, say $K = 256$ and an integer $e \geq 2$. Further, we define $\eta = \omega^{N/J}$, with $J = K^{e-1}$ and assume that multiplying an arbitrary element of $\mathbb{Z}/p\mathbb{Z}$ by $\eta^i$, for any $i = 0, \ldots, K-1$, can be done within $O(k)$ machine-word operations. Consequently, every arithmetic operation (addition, multiplication) involved in a DFT of size $K$, using $\eta$ as a primitive root, amounts to $O(k)$ machine-word operations. Therefore, such DFT of size $K$ can be performed with $O(K\ \log(K)\, k)$ machine-word operations. As we shall see in Chapter 3, this latter result holds whenever $p$ is a so called *generalized Fermat number*.

Considering now a DFT of size $N$ at $\omega$. Using the factorization formula of Cooley and Tukey,

$$\mathrm{DFT}_{JK} \;=\; (\mathrm{DFT}_J \otimes I_K) D_{J,K} (I_J \otimes \mathrm{DFT}_K) L_J^{JK}, \tag{1.10}$$

see Section 2.2, the DFT of $f$ at $\omega$ is essentially performed by:

1. $K^{e-1}$ DFT's of size $K$ (that is, DFT's on polynomials of degree at most $K-1$),
2. $N$ multiplications by a power of $\omega$ (coming from the diagonal matrix $D_{J,K}$) and
3. $K$ DFT's of size $K^{e-1}$.

Unrolling Formula (2.4) so as to replace $\mathrm{DFT}_J$ by $\mathrm{DFT}_K$ and the other linear operators involved (the diagonal matrix $D$ and the permutation matrix $L$) one can deduce that a DFT of size $N = K^e$ reduces to:

1. $e\, K^{e-1}$ DFT's of size $K$, and
2. $(e-1)\, N$ multiplication by a power of $\omega$.

Recall that the assumption on the cost of a multiplication by $\eta^i$, for $0 \leq i < K$, makes the cost for one DFT of size $K$ to $O(K\ \log_2(K)\, k)$ machine-word operations. Hence, all the DFT's of size $K$ together amount to $O(e\, N\ \log_2(K) k)$ machine-word operations, that is, $O(N\ \log_2(N)\, k)$ machine-word operations. Meanwhile, the total cost of the multiplication by a power of $\omega$ is $O(e\, N\, k^2)$ machine-word operations, that is, $O(N\ \log_K(N)\, k^2)$ machine-word operations. Indeed, multiplying an arbitrary element of $\mathbb{Z}/p\mathbb{Z}$ by an arbitrary power of $\omega$ requires a long multiplication at a the cost $O(k^2)$ machine-word operations. Therefore, under our assumption, a DFT of size $N$ at $\omega$ amounts to

$$O(N\ \log_2(N)\, k \;+\; N\ \log_K(N)\, k^2) \tag{1.11}$$

machine-word operations. When using generalized Fermat primes, we have $K = 2k$. Hence, the second term in the big-oh notation, dominates the first one.

Without our assumption, as discussed earlier, the same DFT would run in $O(N\ \log_2(N)\, k^2)$ machine-word operations. Therefore, using generalized Fermat primes brings a speedup factor of $\log(K)$ w.r.t. the direct approach using arbitrary prime numbers.

At this point, it is natural to ask what would be the cost of a comparable computation using small primes and the CRT. To be precise, let us consider the following problem. Let $p_1, \ldots, p_k$ pairwise different prime numbers of machine-word size and let $m$ be their product. Assume that $N$ divides each of $p_1 - 1, \ldots, p_k - 1$ such that the each of fields $\mathbb{Z}/p_1\mathbb{Z}, \ldots, \mathbb{Z}/p_e\mathbb{Z}$ admits a $N$-th primitive roots of unity, $\omega_1, \ldots, \omega_k$. Then, $\omega = (\omega_1, \ldots, \omega_k)$ is an $N$-th primitive root of $\mathbb{Z}/m\mathbb{Z}$. Indeed, the ring $\mathbb{Z}/p_1\mathbb{Z} \otimes \cdots \otimes \mathbb{Z}/p_e\mathbb{Z}$ is a direct product of fields. Let $f \in \mathbb{Z}/m\mathbb{Z}[x]$ be a polynomial of degree $N-1$. One can compute the DFT of $f$ at $\omega$ in three steps:

1. Compute the images $f_1, \ldots, f_k$ of $f$ in $\mathbb{Z}/p_1\mathbb{Z}[x], \ldots, \mathbb{Z}/p_k\mathbb{Z}[x]$.
2. Compute the DFT of $f_i$ at $\omega_i$ in $\mathbb{Z}/p_i\mathbb{Z}[x]$, for $i = 1, \ldots, k$,
3. Combine the results using CRT so as to obtain a DFT of $f$ at $\omega$.

The first and the third above steps will run within $O(k \times N \times k^2)$ machine-word operations meanwhile the the second one amount to $O(k \times N \log(N))$ machine-word operations.

These estimates seem to suggest that the big prime field approach is slower than the small prime fields approach by a factor of $k/\log(K)$. However, we should keep in mind that $k$ and $K$ are small constants meanwhile $N$ is the only quantity which is arbitrary large. Thus, the factor $k/\log(K)$ does not mean much, at least theoretically. Moreover, the big prime field FFT approach and the above second step in the small prime field FFT approach have similar memory access patterns and costs. Indeed, they use the same 6-step FFT algorithm. Hence, the above first and third steps are overheads to the small prime field FFT approach in terms of memory access costs.

Therefore, it is hard to compare the computational efficiency of the two approaches by using theoretical arguments only. In other words, experimentation is needed and this is what this thesis is about.

The contributions of this thesis are as follows:

1. We present algorithms for arithmetic operations in the “big” prime field $\mathbb{Z}/p\mathbb{Z}$, where $p$ is a generalized Fermat number of the form $p = r^k + 1$ where $r$ fits a machine-word and $k$ is a power of 2.
2. We report on an a GPU (Graphics Processing Units) implementation of those algorithms as well as a GPU implementation of an FFT over such big prime field.

3. Our experimental results show that
   (a) computing an FFT of size $N$, over a big prime field for $p$ fitting on $k$ 64-bit machine-words, and
   (b) computing $2k$ FFTs of size $N$, over a small prime field (that is, where the prime fits a 32-bit half-machine-word) followed by a combination (i.e. CRT-like) of those FFTs

   are two competitive approaches in terms of running time. Since the former approach has the benefits mentioned above (in the area of polynomial system solving), we view this experimental observation as a promising result.

The reasons for a GPU implementation are as follows. First, the model of computations and the hardware performance provide interesting opportunities to implement big prime field arithmetic, in particular in terms of vectorization of the program code. Secondly, highly optimized FFTs over small prime fields have been implemented on GPUs by Wei Pan [17, 18] and we use them in our experimental comparison.

This thesis is organized as follows:

- Chapter 2 gathers background materials on GPU programming and FFTs.
- Chapter 3 presents algorithms for performing additions and multiplications in the big prime field $\mathbb{Z}/p\mathbb{Z}$.
- Chapter 4 contains our GPU implementation of the algorithms of Chapter 3.
- Chapter 5 discusses how to efficient implement on GPUs the permutations that are required by FFT algorithms.
- Chapter 6 explains how to take advantage of Coolye-Tukey factorization formula in the context of the trick of Martin Fürer for computing FFTs over the big prime field $\mathbb{Z}/p\mathbb{Z}$. A GPU implementation of those ideas follows.
- Chapter 7 reports on the experimental comparison "big vs small" that was mentioned above.

Chapter 3 is based on a preliminary work by Svyatoslav Covanov, a former student of Professor Marc Moreno Maza. A first GPU implementation of the algorithms in Chapters 3 together with a GPU implementation of FFTs over the big prime field $\mathbb{Z}/p\mathbb{Z}$ was attempted by Dr. Liangyu Chen[1] (a former visiting scholar working with Professor Marc Moreno Maza) but yielded unsatisfactory experimental results.

[1] http://faculty.ecnu.edu.cn/s/187/t/1487/main.jspy

# Chapter 2

# Background

In this chapter, we review the basic principles of GPGPU computing and fast Fourier transforms. First, in Section 2.1, we explain GPGPU computing, and specifically, how we can develop parallel programs in the NVIDIA CUDA programming model. Then, in Section 2.2, we explain *fast Fourier transform* and its related definitions.

## 2.1 GPGPU computing

Parallel programming has always been considered as a difficult task. Among many available platforms, *general purpose graphics processing unit (GPGPU) computing* has proven to be a cost-effective solution for scientific computing. GPUs are parallel processors that can handle huge amounts of data. This makes GPUs the suitable type of platform for data parallel algorithms. *Data-parallelism* refers to a type of computation in which the work can be distributed to lots of smaller tasks, with little or no dependency between them. In less than a decade, GPGPU computing has evolved from a cutting edge technology to one of the mainstream solutions for high-end computing, specifically NVIDIA corporation has played a huge role in developing and promoting the CUDA programming model (see [19] for more details). In this section, we explain preliminary definitions and keywords that will be frequently used in relation to the CUDA programming model. Definitions and examples of this chapter are based on [7] and [8].

### 2.1.1 CUDA programming model

*Compute Unified Device Architecture*, or CUDA, is a programming model and language extension that is developed and supported by NVIDIA corporation. The CUDA platform provides language extensions in C/C++ and a number of other languages. The main purpose of the CUDA platform is to provide a simplified interface for writing scalable parallel programs that can be easily recompiled on GPU cards of different architectures.

**Thread.** A *thread* is the smallest computational unit in the CUDA programming model. At the time of execution, every thread will be assigned to one scalar processor. Also, each thread belongs to a thread block. Finally, each thread has a unique index inside its respective thread block, which depending on dimensions of the thread block can be accessed via

1. threadIdx.x,
2. threadIdx.y (only if the thread belongs to a 2D or 3D thread block),
3. threadIdx.z (only if the thread belongs to a 3D thread block).

**Thread block.** A group of threads together form a *thread block*. Each thread block belongs to a grid. Finally, each thread block has a unique index inside its respective grid, which depending on the dimensions of the grid can be accessed via

1. blockIdx.x,
2. blockIdx.y (only if the thread block block belongs to a 2D or 3D grid),
3. and blockIdx.z (only if the thread block belongs to a 3D grid).

Figure 2.1 illustrates an example of a two dimensional thread block with 2 rows and 6 columns.

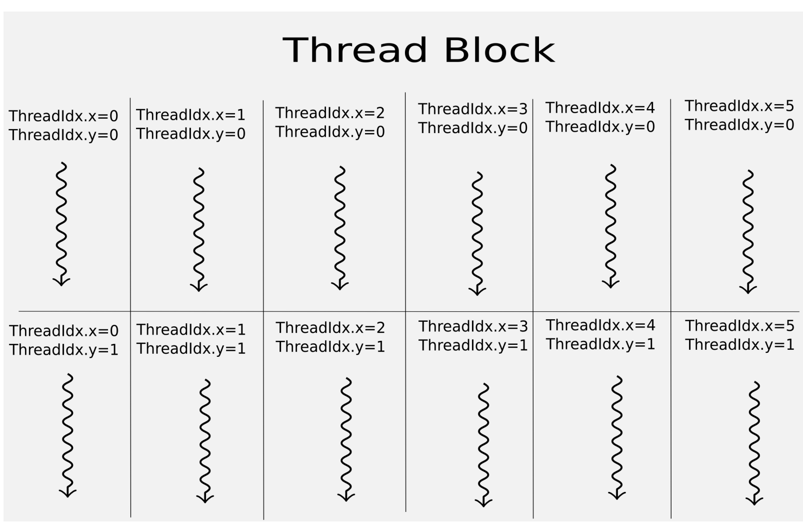


Figure 2.1: Example of a 2D thread block with 2 rows and 6 columns.

**Grid.** A group of independent thread blocks together form a *grid*. CUDA-capable GPUs can support 2D or 3D grids (depending on their architecture). Figure 2.2 illustrates an example of a two dimensional block with 2 rows and 4 columns.

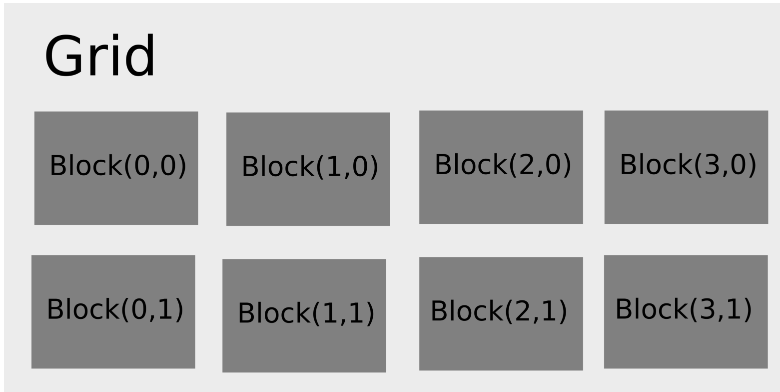


Figure 2.2: Example of a 2D grid with 2 rows and 4 columns.

**Kernel.** At the time of execution, all threads in all thread blocks will run the same function. which is known as *Kernel*.

**Device.** In the CUDA programming model, *device* refers to the GPU that executes kernels on threads.

**Host.** In the CUDA programming model, *host* refers to the CPU that initializes kernels. Figure 2.3 shows the relationship between the host and the device.


Figure 2.3: Host and device in the CUDA programming model.

**Compute capability (CC).** Every CUDA device is built on a core architecture with some specific capabilities. Each device is numbered by a *Compute capability (CC)*, which is of the form $A.B$. This numbering makes it easier to distinguish architectures from each other. In this presentation, $A$ as the major part, specifies the architecture series, and $B$, as the minor part, relates to the special improvements to each architecture. For example, devices of compute capability 3.0, 3.1, and 3.2 have the same architecture core, however, they have different hardware optimizations.

**Warp.** Every 32 threads inside a thread block form a *warp*.

**Streaming multiprocessor.** *Streaming multiprocessors* (SMs) are building blocks of GPUs. Each streaming multiprocessor has a number of scalar processors, registers, warp schedulers, and cache. At the time of execution, the device driver will assign each thread block to one streaming multiprocessor. After being scheduled by the warp scheduler, each thread of the thread block will run the kernel on one processing core.

**Warp scheduler.** At the time of execution, each streaming multiprocessor partitions threads into warps. In the next step, warps will be scheduled by a *warp scheduler* for execution on scalar processors. Table 2.1 shows the maximum number of warps that can reside on streaming multiprocessors of different compute capabilities.

| Compute capability | 1.0/1.1 | 1.2/1.3 | 2.x | 3.x and higher |
|---|---|---|---|---|
| The maximum number of threads per SM | 768 | 1024 | 1536 | 2048 |
| The maximum number of warps | 24 | 32 | 48 | 64 |

Table 2.1: The maximum number of warps per streaming multiprocessor.

### 2.1.2 CUDA memory model

The CUDA platform has multiple levels of memory. As a programmer, it is critical to use different types of GPU memory properly. In other words, each level of GPU memory should be used for a specific type of application. Figure 2.4 shows levels of GPU memory for devices of compute capability 2.0 and higher.

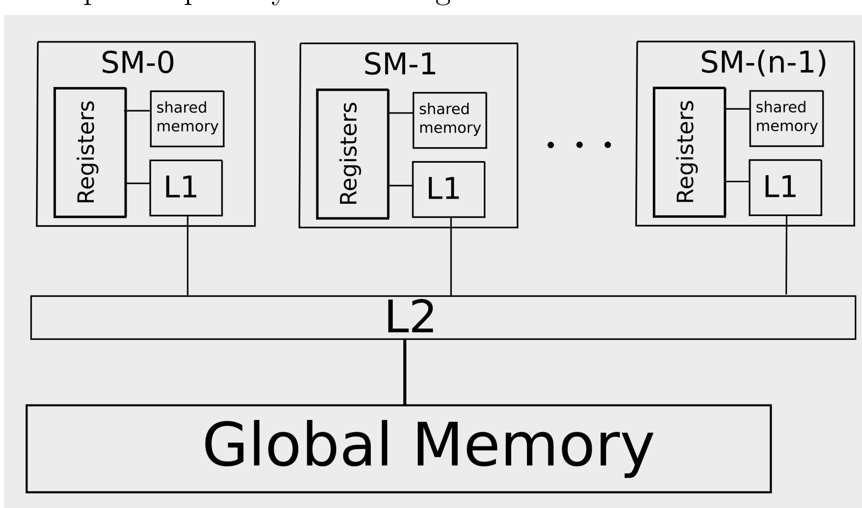


Figure 2.4: CUDA memory hierarchy for CC 2.0 and higher.

**On-chip memory.** This type of memory is located on the streaming multiprocessor. Registers, shared memory, and L1 cache are examples of on-chip memory. All other levels of GPU memory are considered as *off-chip memory*.

**Registers.** Registers are fastest type of memory on GPUs. Accessing to a register has almost no cost, because it is placed on the streaming multiprocessor. Each streaming multiprocessor has a limited number of registers. Table 2.2 shows the number of available registers on one streaming multiprocessor for CUDA-capable NVIDIA GPUs.

| Compute capability | 1.x | 2.x | 3.x | 4.x | 5.x | 6.x |
|---|---|---|---|---|---|---|
| The number of 32-bit registers/SM | 124 | 63 | 255 | 255 | 255 | 255 |

Table 2.2: The number of 32-bit registers per streaming multiprocessor.

**Global memory.** This type of memory is available to all threads in all thread blocks. Global memory is the slowest type of GPU memory.

**Coalesced accesses to global memory.** Inside a warp, consecutive threads can have access to consecutive words in global memory in a *coalesced* way. For doing so, the GPU driver translates multiple read or write memory calls into a single memory call. For current CUDA-enabled GPUs,

**L1 Cache.** This type of on-chip GPU memory is accessible by all threads inside a warp. GPUs have comparably less amount of L1 cache per multiprocessor than CPUs have. Depending on the GPU architecture (CC), the programmer can enable or disable the L1 caching.

**L2 Cache.** This type of off-chip GPU memory is available on devices of compute capability 2.0 and higher. If L1 cache is enabled, all read requests to global memory will first go through the L1 cache, and then through the L2 cache. However, if the L1 cache is disabled, all read transactions will go directly through the L2 cache.

**Local memory.** Each thread can have a private off-chip memory, known as *local memory*. Local memory is allocated on global memory, therefore, accesses to local memory will be slow. However, accesses to local memory will be coalesced if adjacent threads of the same warp will have access to the same index of an array. Devices of compute capability 1.x have 16 KB of local memory. Finally, devices of other compute capabilities have 512 KB of local memory.

**Shared memory.** This type of memory is available to all threads inside the thread block. It can be used

1. for communicating between threads inside the thread block, and
2. as a low cost memory (similar to registers) for storing temporary variables of each thread.

On the positive side, accesses to shared memory have almost no cost, because, compared to registers, it only takes a few more cycles. On the negative side, shared memory accesses can go through bank conflicts, meaning that all accesses will be serialized.

**Constant memory.** This type of read-only memory is accessible to all threads of a grid and can be used for storing constant data. In order to use constant memory efficiently, all accesses should be to the same memory address at the same time. Otherwise, memory requests will be serialized. Currently, the total amount of constant memory for GPUs of all compute capabilities is equal to 64 KB.

**Texture memory.** This is another type of read-only memory and similar to constant memory, can be used for storing constant data. However, unlike constant memory, scattered to constant memory will not be serialized.

### 2.1.3 Examples of programs in CUDA

In this section, we present two simple examples of programming in the CUDA-C/C++.

**Simple vector addition in CUDA.** Figure 2.5 presents a pseudo-code for computing vector addition on GPUs in the following way:

1. First, the program allocates host memory for `host_a`, `host_b` as input array, and for `host_c` as the output array. (L16:L18)
2. The program reads input data from files into `host_a` and `host_b`, respectively (L21:L22).
3. In next step, the program allocates *device* memory for `device_a`, `device_b`, `device_c` (L25:L27).
4. Then, the program copies input vectors from host memory to device memory.
5. The program sets dimensions of the thread block and grid block, respectively (L34 and L37).
6. At this point, the program invokes the CUDA kernel simpleVectorAddition (L40).
7. Now, inside the kernel, each thread computes its index with respect to its thread block index and size of the thread block, and then, it computes the result of addition for two elements of the same relative index from each input array (L4:L5).
8. After completing the computation by the device, the program copies back the result of computation into the output array, `host_c` (L44).

**Naive matrix transposition in CUDA.** In this example, we explain how we can transpose a $16 \times 16$ matrix by using shared memory of GPUs. For an input array of

```
__global__ void simpleVectorAddition
(int* device_a, int* device_b, int* device_c, int n)
{ /* computing the thread index */
 int tid = blockIdx.x*blockDim.x + threadIdx.x;
 if (tid < n) { device_c[tid] = device_b[tid] + device_a[tid];}
}
int main (int argc, char**argv)
{
 /* pointers to host memory */
 int *host_a, *host_b, *host_c;
 /* pointers to device memory */
 int *device_a, *device_b, *device_c;
 /* size of input vector */
 int n = 1024*1024;
 /* allocating arrays on the host memory */
 host_a = (int*)malloc(sizeof(int)*n);
 host_b = (int*)malloc(sizeof(int)*n);
 host_c = (int*)malloc(sizeof(int)*n);
 /* reading input vectors from files a.dat and b.dat, respectively.*/
 host_a = readInputFromFile("a");
 host_b = readInputFromFile("b");
 /* allocating arrays on the device memory */
 cudaMalloc( (void**)&device_a, n*sizeof(int));
 cudaMalloc( (void**)&device_b, n*sizeof(int));
 cudaMalloc( (void**)&device_c, n*sizeof(int));
 /* copy data from the host to the device memory */
 cudaMemcpy(device_a, a, sizeof(int)*n, cudaMemcpyHostToDevice);
 cudaMemcpy(device_b, b, sizeof(int)*n, cudaMemcpyHostToDevice);
 //setting up dimensions of a thread block
 dim3 blockDim = (512,1,1);
 //setting up dimensions of the grid
 dim3 gridDim = (n/blockDim.x,1,1);
 //invoking the kernel from host
 simpleVectorAddition <<< gridDim, blockDim >>>
 (device_a, device_b, device_c,n);
 //copy back the results from device memory to host memory
 cudaMemcpy(host_c, device_c, sizeof(int)*n, cudaMemcpyDeviceToHost);
 return 0;
}
```

Figure 2.5: A CUDA example for computing point-wise addition of two vectors.

```
1 #define BLOCK_SIZE 512
2 // tranposing an array of matrices,
3 // each of size 16x16
4 __global__ void matrix_transposition_16
5 (int* device_x, int* device_y, int n)
6 { /* computing the thread index */
7  int tid = blockIdx.x*blockDim.x + threadIdx.x;
8  __shared__ int sharedMem[BLOCK_SIZE];
9  int total=0;
10 if (tid < n) { sharedMem[threadIdx.x] = device_x[tid];}
11 __syncthreads();
12 if (tid<n){
13  i = threadIdx/16;
14  j = threadIdx % 16;
15  offsetOut= i + 16j;
16  device_y[tid]=sharedMem[offsetOut]; //y(j,i):=x(i,j)
17 }
18 }
```

Figure 2.6: A CUDA example for transposing matrices by using shared memory.

size $n$, our example computes transposition for $n/256$ matrices. We assume that the kernel configuration is similar to that of the previous example. This kernel computes the transposition in the following way.

1. Each thread computes its index with respect to its thread block index and size of the thread block (L7).
2. In the next step, a shared array of size `BLOCK_SIZE` is allocated for all threads of the thread block (L8).
3. Then, each thread reads its corresponding value from the input vector into its respective shared memory address (L10).
4. The barrier **syncthreads**() synchronizes all threads of the thread block (L11).
5. At this step, each thread computes the row number and the column number of its corresponding value in the input vector, namely, $(i, j)$ (L13 and L14).
6. In the next step, each thread computes the offset for its corresponding memory address in output vector, namely, $(j, i)$ (L15).
7. Finally, each thread writes its corresponding value to the output vector (L16).

Notice that this kernel does not result in an efficient transposition, because it will have shared memory bank conflicts. It is only mentioned as an illustrative example.

### 2.1.4 Performance of GPU programs

**Bandwidth.** *Bandwidth* refers to the rate of transferring data between two memory addresses (that might be in different levels). *Theoretical bandwidth* is the maximum value for the GPU memory bandwidth which can be calculated by $B_T = f \times w \times 2$ with

1. $f$ as the clock frequency of the GPU memory, and
2. $w$ as the width of memory interface (in terms of number of bytes).

For example, for a GPU memory with the clock rate of 1 GHZ and the memory interface of 384 bits wide, we have

$$B_T = 1 \times 10^9 \times \frac{384}{8} \times 2 = 96 \text{ GB/s}.$$

**Practical bandwidth.** *Practical (effective) bandwidth* is the bandwidth that can be achieved on a GPU in practice. Practical bandwidth can be computed by

$$B_E = \frac{(d_r + d_w)}{t} \tag{2.1}$$

where

1. $d_r$ is the amount of data that is being read from the memory,
2. $d_w$ is the amount of data that is written to the memory, and
3. $t$ is the elapsed time for reading from the memory and writing to the memory.

For example, if the program spends 4 milliseconds for copying a vector of $N = 2^{20}$ long integers (each of size of 8 machine-words) to another vector, then effective bandwidth is

$$B_E = \frac{((2^{20} \times 8 \times 8) \times 2)}{(4 \times 10^{-3})} = 33.5 \text{ GB/s}.$$

Value of practical bandwidth is always less than the value of theoretical bandwidth. Also, enabling some error correction features (like Error-Correcting-Code in NVIDIA cards) can further reduce the effective bandwidth.

**Occupancy.** Occupancy refers to the ratio of the total number of running warps to the maximum number of warps that can be concurrently executed on each streaming multiprocessor. Following factors can affect the percentage of achieved occupancy:

1. the amount of shared memory per each streaming multiprocessor,
2. the number of registers per each thread,
3. the occurrence of register spilling, and finally,

4. the size of a thread block (which we would prefer to be a multiple of 32).

**Data latency.** This term refers to the time spent between requesting the data by a warp and when the data is ready to be processed by the warp. During this time, the warp scheduler executes another warp, therefore, the requesting warp should be waiting. We try to hide the data latency by increasing the occupancy percentage.

**Register spilling.** As long as there are enough registers left to be allocated, single variables and constant values will always be stored in registers. However, an array inside a thread will not always be stored in registers. In fact, the compiler makes the decision to store an array in registers of the streaming multiprocessor only if the following conditions are met:

- the compiler should be able to determine the indexes of the array, and
- there should be enough number of registers to allocate to the array.

Otherwise, the array will be stored in local memory, which will result in *register spilling*. As we explained before, accesses to local memory is costly, therefore, register spilling will have a negative impact on the memory bandwidth. Also, even if the register spilling does not happen, allocating too many registers to each thread will lower the number of concurrent warps, and consequently, will lower the overall occupancy of the application.

**Shared memory bank conflicts.** Shared memory is divided into partitions of the same size, namely, *shared memory banks*. The default size of a shared memory bank is 32 bits, however, for devices of compute capability 2.0 and higher, size of shared memory banks can be configured to 64 bits. Inside a warp, multiple accesses to the same address of shared memory will result in *shared memory bank conflicts*. As a result, conflicted accesses will be serialized, and therefore, will lower the bandwidth.

**Arithmetic bound kernels.** Arithmetic bound kernels spend most of the computation time for issuing arithmetic instructions. In other words, performance is limited by the high number of arithmetic instructions that should be issued at each clock cycle. For an arithmetic bound kernel, we would prefer to lower warp divergence and therefore, avoid using if-else statements as much as possible. Also, we can balance the computation among arithmetic units of each streaming multiprocessor. For example, we can compute part of the integer arithmetic to the floating point arithmetic units and Special Function Units (SFUs).

**Memory bound kernels.** A kernel is memory bound if it spends most of the time for issuing memory requests. As a result, performance will be limited by memory overheads.

An effective solution for increasing performance of memory bound kernels is to make sure the data latency is minimized and more warps will be concurrently executed. In other words, occupancy should be increased to hide the latency. Also, we must ensure that accesses to global memory are minimized by

1. storing data in a data structure that facilitates coalesced accesses, and
2. (if possible) reusing the same data for more computations.

As a final note, for a memory bound GPU kernel, the practical bandwidth is usually close to the peak of the theoretical bandwidth.

**Arithmetic intensity.** *Arithmetic intensity* is defined as the ratio of the number of arithmetic instructions to the total amount of processed data. More importantly, this term does not have a unique definition. For example, we can define the total amount of processed data

1. as the total number of memory instructions, or
2. as the amount of data in terms of bytes.

**Instruction level parallelism (ILP).** This term refers to the parallelization of independent instructions at the level of hardware. For example, assume that $a_i, b_i, c_i$ $(0 \leq i < 4)$ are pointers to non-overlapping addresses in the memory. Then, as shown in Figure 2.7, we can concurrently compute 4 additions $a_i := b_i + c_i$ by using 4 threads.

| `tid` | 0 | 1 | 2 | 3 |
|---|---|---|---|---|
| Instruction | $a_0 = b_0 + c_0$ | $a_1 = b_1 + c_1$ | $a_2 = b_2 + c_2$ | $a_3 = b_3 + c_3$ |

Figure 2.7: Four independent instructions.

On the other hand, as shown in Figure 2.8, one thread can be used for computing all four additions. However, in practice, it is very difficult to exploit the ILP, mostly because the programmer does not have direct control over it. In fact, it is the compiler that makes the decision for using ILP. Depending on the architecture of the device, 2 or 4 instructions might be parallelized in this way.

| tid | 0 |
|---|---|
| Instruction | $a_0 = b_0 + c_0$<br>$a_1 = b_1 + c_1$<br>$a_2 = b_2 + c_2$<br>$a_3 = b_3 + c_3$ |

Figure 2.8: An example of ILP.

### 2.1.5 Profiling CUDA applications

**Profiler.** A profiler is software that is used for inspecting the performance of an application. As part of the software development kit (CUDA-SDK), NVIDIA corporation provides `nvprof` as the official command-line profiler for CUDA applications. In next step, we explain a number of the most important metrics that can be measured by this profiler. Moreover, Table 2.3 shows a list of the `nvprof` metrics that will be used for measuring the performance of our implementation.

**Instruction per cycle (IPC).** This metric measures the total number of instructions that are issued on each streaming multiprocessor at each clock cycle.

**Achieved occupancy.** This metric represents the ratio of the total number of running warps to the maximum possible number of the warps that can be executed on the multiprocessor.

**Instruction replay overhead.** This metric represents the following ratio:

$$\frac{N_{(issued)} - N_{(requested)}}{N_{(requested)}} \tag{2.2}$$

where:

1. $N_{(issued)}$ is the total number of issued instructions, and
2. $N_{(requested)}$ is the total number of requested instructions.

There are similar ”replay overhead” metrics for some other instructions, for example, global memory replay overhead and shared memory replay overhead measure overheads of global memory and shared memory instructions, respectively.

**Global memory load and store throughput.** This metric measures the throughput for all global memory load and store transactions, including accesses to the L1 cache and to the L2 cache.

**DRAM read and write throughput.** This metric measures the memory throughput for memory read transactions between the device memory and the L2 cache.

| Metric name | description |
|---|---|
| `achieved_occupancy` | Percentage of occupancy for all SMs |
| `ipc` | Instruction per cycle |
| `gst_throughput` | Global memory store throughput |
| `gld_throughput` | Global memory load throughput |
| `gst_efficiency` | Global memory store efficiency |
| `dram_utilization` | Device memory utilization (a value between 0 and 10) |

Table 2.3: A short list of performance metrics of `nvprof`.

### 2.1.6 A note on psuedo-code.

We present our algorithms in pseudo-codes similar to the CUDA programming model.

***Host* functions.** Name of this type of function begins with the keyword Host. Host functions can only be called from the host (CPU). Moreover, this type of function are used for

1. initializing the input data, and
2. invoking GPU kernels.

***Kernel* functions.** The name of this type of function begins with the keyword Kernel. Kernel functions will be loaded on each streaming multiprocessor, then, all threads will execute the same code. Kernel functions can only be called from host functions. Finally, this type of function never returns any values, instead, they only depend on global memory for communicating to the host.

***Device* functions.** The name of this type of function begins with the keyword Device. Device functions can only be called from kernel functions. However, device functions can return values to their invoker kernel.

**Size of a machine-word.** We assume that a machine-word (register) is 64-bits wide.

**Fortran style arrays.** In this thesis, we present arrays in the following way:

1. $\vec{\mathrm{x}}$ refers to vector of digits, each of size of of a machine-word,
2. $\vec{\mathrm{x}}[i]$ refers to $i$-th digit of $\vec{x}$, and
3. $\vec{\mathrm{x}}[i : j]$ refers to $i$-th, …, $j$-th digits of $\vec{x}$.

## 2.2 Fast Fourier Transforms

In this section, we review the Discrete Fourier Transform over a finite field, and its related concepts.

**Primitive and principal roots of unity.** Let $\mathcal{R}$ be a commutative ring with units. Let $N > 1$ be an integer. An element $\omega \in \mathcal{R}$ is a *primitive* $N$-th root of unity if for $1 < k \leq N$ we have $\omega^k = 1 \iff k = N$. The element $\omega \in \mathcal{R}$ is a *principal* $N$-th root of unity if $\omega^N = 1$ and for all $1 \leq k < N$ we have

$$\sum_{j=0}^{N-1} \omega^{jk} = 0. \tag{2.3}$$

In particular, if $N$ is a power of 2 and $\omega^{N/2} = -1$, then $\omega$ is a principal $N$-th root of unity. The two notions coincide in fields of characteristic 0. For integral domains every primitive root of unity is also a principal root of unity. For non-integral domains, a principal $N$-th root of unity is also a primitive $N$-th root of unity unless the characteristic of the ring $\mathcal{R}$ is a divisor of $N$.

**The discrete Fourier transform (DFT).** Let $\omega \in \mathcal{R}$ be a principal $N$-th root of unity. The *$N$-point DFT at $\omega$* is the linear function, mapping the vector $\vec{a} = (a_0, \ldots, a_{N-1})^T$ to $\vec{b} = (b_0, \ldots, b_{N-1})^T$ by $\vec{b} = \Omega\vec{a}$, where $\Omega = (\omega^{jk})_{0\leq j,k\leq N-1}$. If $N$ is invertible in $\mathcal{R}$, then the $N$-point DFT at $\omega$ has an inverse which is $1/N$ times the $N$-point DFT at $\omega^{-1}$.

**The fast Fourier transform.** Let $\omega \in \mathcal{R}$ be a principal $N$-th root of unity. Assume that $N$ can be factorized to $JK$ with $J, K > 1$. Recall Cooley-Tukey factorization formula [6]

$$\mathrm{DFT}_{JK} \;=\; (\mathrm{DFT}_J \otimes I_K) D_{J,K} (I_J \otimes \mathrm{DFT}_K) L_J^{JK}, \tag{2.4}$$

where, for two matrices $A, B$ over $\mathcal{R}$ with respective formats $m \times n$ and $q \times s$, we denote by $A \otimes B$ an $mq \times ns$ matrix over $\mathcal{R}$ called the tensor product of $A$ by $B$ and defined by

$$A \otimes B = [a_{k\ell} B]_{k,\ell} \quad \text{with} \quad A = [a_{k\ell}]_{k,\ell} \tag{2.5}$$

In the above formula, $\mathrm{DFT}_{JK}$, $\mathrm{DFT}_J$ and $\mathrm{DFT}_K$ are respectively the $N$-point DFT at $\omega$, the $J$-point DFT at $\omega^K$ and the $K$-point DFT at $\omega^J$. The *stride permutation matrix* $L_J^{JK}$ permutes an input vector $\mathbf{x}$ of length $JK$ as follows

$$\mathbf{x}[iJ + j] \mapsto \mathbf{x}[jJ + i], \tag{2.6}$$

for all $0 \leq j < J$, $0 \leq i < K$. If $\mathbf{x}$ is viewed as an $K \times J$ matrix, then $L_J^{JK}$ performs a transposition of this matrix. The *diagonal twiddle matrix* $D_{J,K}$ is defined as

$$D_{J,K} = \bigoplus_{j=0}^{J-1} \operatorname{diag}(1, \omega^j, \ldots, \omega^{j(K-1)}), \tag{2.7}$$

Formula (2.4) implies various divide-and-conquer algorithms for computing DFTs efficiently, often refered as fast Fourier transforms (FFTs). See the seminal papers [20] and [11] by the authors of the SPIRAL abd FFTW projects, respectively. This formula also implies that, if $K$ divides $J$, then all involved multiplications are by powers of $\omega^K$.

In the factorization of the matrix $\mathrm{DFT}_{JK}$, viewing the size $K$ as a base case and assuming that $J$ is a power of $K$, Formula (2.4) translates into Algorithm 2.1. In this algorithm, as in the sequel of this section, $\omega \in \mathcal{R}$ be a principal $N$-th root of unity and $(\alpha_0\alpha_1...\alpha_{N-1})$ is a vector whose coefficients are in $\mathcal{R}$.

**Algorithm 2.1** Radix $K$ Fast Fourier Transform in $\mathcal{R}$

```
procedure FFT_{radix K}((α_0 α_1 ... α_{N−1}), ω, N = J · K)
    for 0 ≤ j < J do                                  ▷ Data transposition
        for 0 ≤ k < K do
            γ[j][k] := α_{kJ+j}
        end for
    end for
    for 0 ≤ j < J do                                  ▷ Base case FFTs
        c[j] := FFT_{base−case}(γ[j], ω^J, K)
    end for
    for 0 ≤ k < K do                                  ▷ Twiddle factor multiplication
        for 0 ≤ j < J do
            δ[k][j] := c[j][k] ∗ ω^{jk}
        end for
    end for
    for 0 ≤ k < K do                                  ▷ Recursive calls
        δ[k] = FFT_{radix K}(δ[k], ω^K, J)
    end for
    for 0 ≤ k < K do                                  ▷ Data transposition
        for 0 ≤ j < J do
            α[jK + k] := δ[k][j]
        end for
    end for
    return (α_0 α_1 ... α_{N−1})
end procedure
```

The recursive formulation of Algorithm 2.1 is not appropriate for generating code targeting many-core GPU-like architectures for which, formulating algorithms iteratively facilates the division of the work into kernel calls and thread-blocks.

To this end, we shall unroll Formula (2.4). This will be done in Chapter 6.

# Chapter 3

# Arithmetic Computations Modulo Sparse Radix Generalized Fermat Numbers

The $n$-th Fermat number, denoted by $F_n$, is given by $F_n = 2^{2^n} + 1$. This sequence plays an important role in number theory and, as mentioned in the introduction, in the development of asymptotically fast algorithms for integer multiplication [21, 13].

Arithmetic operations modulo a Fermat number are simpler than modulo an arbitrary positive integer. In particular 2 is a $2^{n+1}$-th primitive root of unity modulo $F_n$. Unfortunately, $F_4$ is the largest Fermat number which is known to be prime. Hence, when computations require the coefficient ring be a field, Fermat numbers are no longer interesting. This motivates the introduction of other family of Fermat-like numbers, see, for instance, Chapter 2 in the text book *Guide to elliptic curve cryptography* [14].

Numbers of the form $a^{2^n} + b^{2^n}$ where $a > 1$, $b \geq 0$ and $n \geq 0$ are called *generalized Fermat numbers*. An odd prime $p$ is a generalized Fermat number if and only if $p$ is congruent to 1 modulo 4. The case $b = 1$ is of particular interest and, by analogy with the ordinary Fermat numbers, it is common to denote the generalized Fermat number $a^{2^n} + 1$ by $F_n(a)$. So 3 is $F_0(2)$. We call $a$ the *radix* of $F_n(a)$. Note that, Landau's fourth problem asks if there are infinitely many generalized Fermat primes $F_n(a)$ with $n > 0$.

In the finite ring $\mathbb{Z}/F_n(a)\mathbb{Z}$, the element $a$ is a $2^{n+1}$-th primitive root of unity. However, when using binary representation for integers on a computer, arithmetic operations in $\mathbb{Z}/F_n(a)\mathbb{Z}$ may not be as easy to perform as in $\mathbb{Z}/F_n\mathbb{Z}$. This motivates the following.

**Definition 1** *We call* sparse radix generalized Fermat number*, any integer of the form $F_n(r)$ where $r$ is either $2^w + 2^u$ or $2^w - 2^u$, for some integers $w > u \geq 0$. In the former case, we denote $F_n(r)$ by $F_n^+(w,u)$ and in the latter by $F_n^-(w,u)$.*

Table 3.1 lists a few sparse radix generalized Fermat numbers (SRGFNs, for short) that are prime. For each $p$ among those numbers, we give the largest power of 2 dividing $p-1$, that is, the maximum length $N$ of a vector to which a radix-$K$ FFT algorithm (like Algorithm 2.1) where $K$ is an appropriate power of 2.

| $p$ | $\max\{2^e \text{ s.t. } 2^e \mid p-1\}$ |
|---|---|
| $(2^{63}+2^{53})^2+1$ | $2^{106}$ |
| $(2^{64}-2^{50})^4+1$ | $2^{200}$ |
| $(2^{63}+2^{34})^8+1$ | $2^{272}$ |
| $(2^{62}+2^{36})^{16}+1$ | $2^{576}$ |
| $(2^{62}+2^{56})^{32}+1$ | $2^{1792}$ |
| $(2^{63}-2^{40})^{64}+1$ | $2^{2500}$ |
| $(2^{64}-2^{28})^{128}+1$ | $2^{3584}$ |

Table 3.1: SRGFNs of practical interest.

**Notation 1** *In the sequel of this section, we consider $p = F_n(r)$, a fixed SRGFN. We denote by $2^e$ the largest power of 2 dividing $p-1$ and we define $k = 2^n$, so that $p = r^k + 1$ holds.*

As we shall see in the sequel of this section, for any positive integer $N$ which is a power of 2 such that $N$ divides $p-1$, one can find an $N$-th primitive root of unity $\omega \in \mathbb{Z}/p\mathbb{Z}$ such that multiplying an element $a \in \mathbb{Z}/p\mathbb{Z}$ by $\omega^{i(N/2k)}$ for $0 \leq i < 2k$ can be done in linear time w.r.t. the bit size of $a$. Combining this observation with an appropriate factorization of the DFT transform on $N$ points over $\mathbb{Z}/p\mathbb{Z}$, we obtain an efficient FFT algorithm over $\mathbb{Z}/p\mathbb{Z}$.

## 3.1 Representation of $\mathbb{Z}/p\mathbb{Z}$

We represent each element $x \in \mathbb{Z}/p\mathbb{Z}$ as a vector $\vec{x} = (x_{k-1}, x_{k-2}, \ldots, x_0)$ of length $k$ and with non-negative integer coefficients such that we have

$$x \equiv x_{k-1}\, r^{k-1} + x_{k-2}\, r^{k-2} + \cdots + x_0 \mod p. \tag{3.1}$$

This representation is made unique by imposing the following constraints

1. either $x_{k-1} = r$ and $x_{k-2} = \cdots = x_1 = 0$,

2. or $0 \leq x_i < r$ for all $i = 0, \ \ldots, \ (k-1)$.

We also map $x$ to a univariate integer polynomial $f_x \in \mathbb{Z}[T]$ defined by $f_x = \sum_{i=0}^{k-1} x_i t^i$ such that $x \equiv f_x(r) \mod p$.

Now, given a non-negative integer $x < p$, we explain how the representation $\vec{x}$ can be computed. The case $x = r^k$ is trivially handled, hence we assume $x < r^k$. For a non-negative integer $z$ such that $z < r^{2^i}$ holds for some positive integer $i \leq n = \log_2(k)$, we denote by $\mathrm{vec}(z, i)$ the unique sequence of $2^i$ non-negative integers $(z_{2^i-1}, \ldots, z_0)$ such that we have $0 \leq z_j < r$ and $z = z_{2^i-1} r^{2^i-1} + \cdots + z_0$. The sequence $\mathrm{vec}(z, i)$ is obtained as follows:

1. if $i = 1$, we have $\mathrm{vec}(z, i) = (q, s)$,
2. if $i > 1$, then $\mathrm{vec}(z, i)$ is the concatenation of $\mathrm{vec}(q, i-1)$ followed by $\mathrm{vec}(s, i-1)$,

where $q$ and $s$ are the quotient and the remainder in the Euclidean division of $z$ by $r^{2^{i-1}}$. Clearly, $\mathrm{vec}(x, n) = \vec{x}$ holds.

We observe that the sparse binary representation of $r$ facilitates the Euclidean division of an non-negative integer $z$ by $r$, when performed on a computer. Referring to the notations in Definition 1, let us assume that $r$ is $2^w + 2^u$, for some integers $w > u \geq 0$. (The case $2^w - 2^u$ would be handled in a similar way.) Let $z_{\mathrm{high}}$ and $z_{\mathrm{low}}$ be the quotient and the remainder in the Euclidean division of $z$ by $2^w$. Then, we have

$$z \;=\; 2^w\, z_{\mathrm{high}} + z_{\mathrm{low}} \;=\; r\, z_{\mathrm{high}} + z_{\mathrm{low}} - 2^u z_{\mathrm{high}}. \tag{3.2}$$

Let $s = z_{\mathrm{low}} + -2^u z_{\mathrm{high}}$ and $q = z_{\mathrm{high}}$. Three cases arise:

(S1) if $0 \leq s < r$, then $q$ and $s$ are the quotient and remainder of $z$ by $r$,

(S2) if $r \leq s$, then we perform the Euclidean division of $s$ by $r$ and deduce the desired quotient and remainder,

(S3) if $s < 0$, then $(q, s)$ is replaced by $(q + 1, s + r)$ and we go back to Step (S1).

Since the binary representations of $r^2$ can still be regarded as sparse, a similar procedure can be done for the Euclidean division of an non-negative integer $z$ by $r^2$. For higher powers of $r$, we believe that Montgomery algorithm is the way go, though this remains to be explored.

## 3.2 Finding primitive roots of unity in $\mathbb{Z}/p\mathbb{Z}$

**Notation 2** *Let $N$ a power of 2, say $2^\ell$, dividing $p-1$ and let $g \in \mathbb{Z}/p\mathbb{Z}$ be a $N$-th primitive root of unity.*

Recall that such an $N$-th primitive root of unity can be obtained by a simple probabilistic procedure. Write $p = qN + 1$. Pick a random $\alpha \in \mathbb{Z}/p\mathbb{Z}$ and let $\omega = \alpha^q$. Little Fermat theorem implies that either $\omega^{N/2} = 1$ or $\omega^{N/2} = -1$ holds. In the latter case, $\omega$ is an $N$-th primitive root of unity. In the former, another random $\alpha \in \mathbb{Z}/p\mathbb{Z}$ should be considered. In our various software implementation of finite field arithmetic [16, 3, 15], this procedure finds an $N$-th primitive root of unity after a few tries and has never been a performance bottleneck.

In the following, we consider the problem of finding an $N$-th primitive root of unity $\omega$ such that $\omega^{N/2k} = r$ holds. The intention is to speed up the portion of FFT computation that requires to multiply elements of $\mathbb{Z}/p\mathbb{Z}$ by powers of $\omega$.

**Proposition 1** *In $\mathbb{Z}/p\mathbb{Z}$, the element $r$ is a $2k$-th primitive root of unity. Moreover, the following algorithm computes an $N$-th primitive root of unity $\omega \in \mathbb{Z}/p\mathbb{Z}$ such that we have $\omega^{N/2k} = r$ in $\mathbb{Z}/p\mathbb{Z}$.*

**Algorithm 3.1** Primitive $N$-th root $\omega \in \mathbb{Z}/p\mathbb{Z}$ s.t. $\omega^{N/2k} = r$

```
procedure PrimitiveRootAsRootOf(N, r, k, g)
    α := g^{N/2k}
    β := α
    j := 1
    while β ≠ r do
        β := αβ
        j := j + 1
    end while
    ω := g^j
    return (ω)
end procedure
```

**Proof** Since $g^{N/2k}$ is a $2k$-th root of unity, it is equal to $r^{i_0}$ (modulo $p$) for some $0 \le i_0 < 2k$ where $i_0$ is odd. Let $j$ be an non-negative integer. Observe that we have

$$g^{j2^\ell/2k} = (g^i\, g^{2\,k\,q})^{2^\ell/2k} = g^{i2^\ell/2k} = r^{i\ i_0}, \tag{3.3}$$

where $q$ and $i$ are quotient and the remainder of $j$ in the Euclidean division by $2k$. By definition of $g$, the powers $g^{i2^\ell/2k}$, for $0 \leq i < 2k$, are pairwise different. It follows from Formula (3.3) that the elements $r^{i\ i_0}$ are pairwise different as well, for $0 \leq i < 2k$. Therefore, one of those latter elements is $r$ itself. Hence, we have $j_1$ with $0 \leq j_1 < 2k$ such that $g^{j_1 N/2k} = r$. Then, $\omega = g^{j_1}$ is as desired and Algorithm 3.1 computes it. $\square$

## 3.3 Addition and subtraction in $\mathbb{Z}/p\mathbb{Z}$

Let $x, y \in \mathbb{Z}/p\mathbb{Z}$ represented by $\vec{x}, \vec{y}$, see Section 3.1 for this latter notation. Algorithm 3.2 computes the representation $\overrightarrow{x+y}$ of the element $(x+y) \mod p$.

**Algorithm 3.2** Computing $x + y \in \mathbb{Z}/p\mathbb{Z}$ for $x, y \in \mathbb{Z}/p\mathbb{Z}$

**procedure** BigPrimeFieldAddition($\vec{x}, \vec{y}, r, k$)

1: compute $z_i = x_i + y_i$ in $\mathbb{Z}$, for $i = 0, \ldots, k-1$,

2: let $c_0 = 0$ and $z_k = 0$,

3: for $i = 0, \ldots, k-1$, compute the quotient $q_i$ and the remainder $s_i$ in the Euclidean division of $z_i$ by $r$, then replace $(z_{i+1}, z_i)$ by $(z_{i+1} + q_i, s_i)$,

4: if $z_k = 0$ then return $(z_{k-1}, \ldots, z_0)$,

5: if $z_k = 1$ and $z_{k-1} = \cdots = z_0 = 0$, then let $z_{k-1} = r$ and return $(z_{k-1}, \ldots, z_0)$,

6: let $i_0$ be the smallest index, $0 \leq i_0 \leq k-1$, such that $z_{i_0} \neq 0$, then let $z_{i_0} = z_{i_0} - 1$, let $z_0 = \cdots = z_{i_0-1} = r - 1$ and return $(z_{k-1}, \ldots, z_0)$.

**end procedure**

**Proof** At Step (1), $\vec{x}$ and $\vec{y}$, regarded as vectors over $\mathbb{Z}$, are added component-wise. At Steps (2) and (3), the carry, if any, is propagated. At Step (4), there is no carry beyond the leading digit $z_{k-1}$, hence $(z_{k-1}, \ldots, z_0)$ represents $x + y$. Step (5) handles the special case where $x + y = p - 1$ holds. Step (6) is the *overflow* case which is handled by subtracting $1 \mod p$ to $(z_{k-1}, \ldots, z_0)$, finally producing $\overrightarrow{x+y}$. $\square$

A similar procedure computes the vector $\overrightarrow{x-y}$ representing the element $(x - y) \in \mathbb{Z}/p\mathbb{Z}$. Recall that we explained in Section 3.1 how to perform the Euclidean divisions at Step (S3) in a way that exploits the sparsity of the binary representation of $r$.

In practice, the binary representation of the radix $r$ fits a machine word, see Table 3.1. Consequently, so does each of the "digit" in the representation $\vec{x}$ of every element $x \in \mathbb{Z}/p\mathbb{Z}$. This allows us to exploit machine arithmetic in a sharper way. In particular, the Euclidean divisions at Step (S3) can be further optimized.

## 3.4 Multiplication by a power of $r$ in $\mathbb{Z}/p\mathbb{Z}$

Before considering the multiplication of two arbitrary elements $x, y \in \mathbb{Z}/p\mathbb{Z}$, we assume that one of them, say $y$, is a power of $r$, say $y = r^i$ for some $0 < i < 2k$. Note that the cases $i = 0 = 2k$ are trivial. Indeed, recall that $r$ is a $2k$-th primitive root of unity in $\mathbb{Z}/p\mathbb{Z}$. In particular, $r^k = -1$ in $\mathbb{Z}/p\mathbb{Z}$. Hence, for $0 < i < k$, we have $r^{k+i} = -r^i$ in $\mathbb{Z}/p\mathbb{Z}$. Thus, let us consider first the case where $0 < i < k$ holds. We also assume $0 \le x < r^k$ holds in $\mathbb{Z}$, since the case $x = r^k$ is easy to handle. From Equation (3.1) we have:

$$\begin{array}{rcl} x r^i & \equiv & x_{k-1}\, r^{k-1+i} + \cdots + x_0\, r^i \quad \bmod p \\ & \equiv & \sum_{j=0}^{j=k-1} x_j r^{j+i} \quad \bmod p \\ & \equiv & \sum_{h=i}^{h=k-1+i} x_{h-i} r^h \quad \bmod p \\ & \equiv & \sum_{h=i}^{h=k-1} x_{h-i} r^h - \sum_{h=k}^{h=k-1+i} x_{h-i} r^{h-k}. \quad \bmod p \end{array}$$

The case $k < i < 2k$ can be handled similarly. Also, in the case $i = k$ we have $xr^i = -x$ in $\mathbb{Z}/p\mathbb{Z}$. It follows, that for all $0 < i < 2k$, computing the product $x\, r^i$ simply reduces to computing a subtraction This fact, combined with Proposition 1, motivates the development of FFT algorithms over $\mathbb{Z}/p\mathbb{Z}$.

## 3.5 Multiplication in $\mathbb{Z}/p\mathbb{Z}$

Let again $x, y \in \mathbb{Z}/p\mathbb{Z}$ represented by $\vec{x}, \vec{y}$ and consider the univariate polynomials $f_s, f_y \in \mathbb{Z}[T]$ associated with $x, y$; see Section 3.1 for this notation. To compute the product $x\, y$ in $\mathbb{Z}/p\mathbb{Z}$, we proceed as follows.

**Algorithm 3.3** Computing $xy \in \mathbb{Z}/p\mathbb{Z}$ for $x, y \in \mathbb{Z}/p\mathbb{Z}$

**procedure** BigPrimeFieldMultiplication$(f_x, f_y, r, k)$

1: We compute the polynomial product $f_u = f_x f_y$ in $\mathbb{Z}[T]$ modulo $T^k + 1$.

2: Writing $f_u = \sum_{i=0}^{k-1} u_i T^i$, we observe that for all $0 \le i \le k-1$ we have $0 \le u_i \le kr^2$ and compute a representation $\overrightarrow{u_i}$ of $u_i$ in $\mathbb{Z}/p\mathbb{Z}$ using the method explained in Section 3.1.

3: We compute $u_i r^i$ in $\mathbb{Z}/p\mathbb{Z}$ using the method of Section 3.4.

4: Finally, we compute the sum $\sum_{i=0}^{k-1} u_i r^i$ in $\mathbb{Z}/p\mathbb{Z}$ using Algorithm 3.2.

**end procedure**

For large values of $k$, $f_x f_y \mod T^k + 1$ in $\mathbb{Z}[T]$ can be computed by asymptotically fast algorithms (see the paper [4]). However, for small values of $k$ (say $k \leq 8$), using plain multiplication is reasonable.

# Chapter 4

# Big Prime Field Arithmetic on GPUs

This chapter describes our CUDA implementation of the algorithms of Chapter 3. In the sequel, $p$ is a sparse radix generalized Fermat number, (see Definition 1) given as $p = r^k + 1$ where

1. $r$ is either $2^w + 2^u$ or $2^w - 2^u$, for some integers $w > u \geq 0$; moreover, the binary representation of $r$ fits within a machine-word,
2. $k$ is a power of 2, namely $k = 2^n$, for a positive $n$.

Our test-examples use $p = (2^{63} + 2^{34})^8 + 1$, thus, $k = 8$ and $r = (2^{63} + 2^{34})$.

In Section 4.1, we explain principles of computing arithmetic operations in $\mathbb{Z}/p\mathbb{Z}$ on GPUs. Then, we explain how we store elements of $\mathbb{Z}/p\mathbb{Z}$ in the GPU memory. In the same section, we explain the impact of different levels of GPU memory on the overall performance of our implementation. Moreover, we describe how transposing the input vector can facilitate coalesced accesses to the memory. Then, in Section 4.2, we explain the general structure of our kernels, also, we present algorithms for computing arithmetic in $\mathbb{Z}/p\mathbb{Z}$ on GPUs. Finally, in Section 4.3, we explain profiling results for our CUDA implementation of arithmetic operations in $\mathbb{Z}/p\mathbb{Z}$.

## 4.1 Preliminaries

In this section, we explain how we can use GPUs for faster computation of arithmetic operations in $\mathbb{Z}/p\mathbb{Z}$. Also, we describe how different levels of GPU memory can affect the performance. Finally, we explain how transposing the input data can minimize memory overheads.

### 4.1.1 Parallelism for arithmetic in $\mathbb{Z}/p\mathbb{Z}$

We have the following possibilities for parallelizing arithmetic in $\mathbb{Z}/p\mathbb{Z}$.

**Data parallelism.** Computing the same operation on elements of an array can be done in parallel, which is known as *data parallelism*. Specifically, computing any component-wise arithmetic operations on vectors over $\mathbb{Z}/p\mathbb{Z}$ can be considered as a data parallel problem.

**Parallelizing arithmetic operations.** A higher degree of parallelism can be achieved if one arithmetic operation can be computed by using more than one thread. It is difficult to efficiently parallelize addition and subtraction over $\mathbb{Z}/p\mathbb{Z}$, simply because these operations might need to propagate carry during the intermediate computation. Also, the same reasoning applies to multiplication by powers of $r$, which basically is computed by a simple data movement followed by one addition and one subtraction. However, as we will see in Section 4.2.5, even though multiplications in $\mathbb{Z}/p\mathbb{Z}$ can be computed in parallel, at the end this parallelization will have frequent accesses to the memory, and therefore, will not improve the overall performance.

**Instruction level parallelism (ILP).** As we explained in Section 2.1.4, at the lowest level, it is possible to exploit parallelism that is provided by hardware instructions. However, if we cannot efficiently parallelize arithmetic operations over $\mathbb{Z}/p\mathbb{Z}$ by multiple threads, in the same way, we also cannot parallelize them by using ILP.

Conclusively, we focus on computing arithmetic operations over $\mathbb{Z}/p\mathbb{Z}$ as a data-parallel problem. In other words, GPU implementation of arithmetic operations will result in memory bound kernels. For that purpose, as we explained in Section 2.1.4, it is crucial to improve the efficiency of memory transactions in order to hide the data latency. Finally, we assume that in every arithmetic operation over $\mathbb{Z}/p\mathbb{Z}$, one thread will compute one element of the final result.

### 4.1.2 Representing data in $\mathbb{Z}/p\mathbb{Z}$

Every element of $\mathbb{Z}/p\mathbb{Z}$ can be represented by a vector of $k$ *digits* of machine-word size. Our GPU functions work on a batch of elements of $\mathbb{Z}/p\mathbb{Z}$. To be precise, such functions take one or more vectors of $N$ elements of $\mathbb{Z}/p\mathbb{Z}$. Thus, the memory space for each of those vectors is $kN$ machine-words. In practice, $N$ is supposed to be a power of 2 such that $N \geq 2^8$.

For a vector $\vec{X}$ of $N$ elements of $\mathbb{Z}/p\mathbb{Z}$ and an non-negative integer $j$ with $0 \leq j < N$, we denote by $\vec{X}_j$ or $\vec{X}[j]$ (depending on the context) the $j$-th element of $\vec{X}$. Moreover, $\vec{X}_{(j,i)}$ represents the $i$-th digit of the $j$-th element of $\vec{X}$, for $0 \leq i < k$. Therefore, for $0 \leq i < k$ and $0 \leq j < N$ we have:

$$\vec{X}_j = (\vec{X}_{(j,0)}, \ldots, \vec{X}_{(j,k-1)}).$$

Note that this representation is independent from the way that the elements of $\vec{X}$ are stored in the memory.

In the rest of this section, we explain the following concerns with the memory:

- location of data in the memory, and
- minimizing memory overheads.

### 4.1.3 Location of data

As we explained in Section 2.1.2, GPUs have multiple levels of memory, and each level should be used for a specific type of application. At the same time, we must take into account that each streaming multiprocessor has a limited number of on-chip resources, such as the number of registers and the amount of shared memory for each thread block.

By this assumptions, we explain the impact of the following levels of GPU memory on the overall performance.

**Registers.** Using registers can lower the memory efficiency in the following ways:

1. register spilling will increase the data latency (see Section 2.1.4), and
2. using too many registers per thread can lower the occupancy percentage.

It is highly possible that register spilling will happen. For example, some arithmetic operations (e.g. the multiplication by powers of $r$) need to store multiple temporary arrays of $k$ digits, which depending on the value of $k$, cannot be stored in registers. In this case, part of the array will be stored in registers, while the rest of it will be moved to local memory. Consequently, the performance is lowered because of the low occupancy and the high data latency. However, by limiting the maximum number of registers per thread, we guarantee that the excessive amount of data will *always* be stored in local memory of each thread. That is, all threads will use register to an extent that does not lower the occupancy, and therefore, performance will only be lowered due to register spilling. For example, assume that we have a device that has:

- the total number of $64K$ (65536) registers per streaming multiprocessor,
- the maximum number of 64 resident warps per streaming multiprocessor, and
- the total number of 4 streaming multiprocessors.

Therefore, this device can schedule $4 * 64$ active warps for execution. Assume that a kernel, say `d1`, uses 20 registers per thread, while another kernel, say `d2`, uses 60 registers per thread. Therefore:

- For `d1`, $\frac{64K \text{ registers}}{20} \times \frac{32 \text{ threads}}{1 \text{ warp}} = 102$ warps will be scheduled. Therefore, the occupancy percentage will be $102/256 = 39\%$.

- For `d2`, $\frac{64K \text{ registers}}{60} \times \frac{32 \text{ threads}}{1 \text{ warp}} = 34$ warps will be scheduled. Therefore, the occupancy percentage will be $68/256 = 13\%$.

Now, if both `d1` and `d2` have the same effective read and write bandwidth, we would prefer to have `d1` as our implementation.

**Shared memory.** This level of memory can be used in the following ways:

1. for sharing data among threads of a thread block (which is not the case for arithmetic over $\mathbb{Z}/p\mathbb{Z}$), or
2. as a user-managed cache for storing temporary data.

In the latter case, we must take into account that there is a limited amount of shared memory on each streaming multiprocessor. Therefore, for $S$ bytes of shared memory on each streaming multiprocessor, we cannot store more than $\frac{S}{8k}$ elements of $\mathbb{Z}/p\mathbb{Z}$ per thread block. Hence:

1. for larger values of $k > 16$, shared memory cannot be used for storing all digits of one element of $\mathbb{Z}/p\mathbb{Z}$, and
2. for smaller values of $k$ ($8 \leq k \leq 16$), using shared memory will lower the occupancy percentage.

Conclusively, we would prefer to avoid using shared memory for computing arithmetic operations, and later, for computing FFTs over $\mathbb{Z}/p\mathbb{Z}$.

**Texture memory.** As we explained in Section 2.1.2, texture memory is used for storing read-only arrays, also, different addresses of can be accessed at the same time (scattered access). As we will explain in Chapter 6, using texture memory can only be useful for computing some multiplications in FFT over $\mathbb{Z}/p\mathbb{Z}$.

**Constant memory.** As we explained in Section 2.1.2, scattered accesses to constant memory will be serialized. There are no opportunities to use constant memory for computing any of the arithmetic operations, and later, for computing the FFT over $\mathbb{Z}/p\mathbb{Z}$.

Finally, we keep whole input data on global memory, and we avoid all other levels of memory on a GPU. In the rest of this section, we focus on improving the efficiency of global memory transactions for computing arithmetic operations in $\mathbb{Z}/p\mathbb{Z}$.

### 4.1.4 Transposing input data

We should use a data structure that facilitates coalesced accesses to global memory. For an input vector $\vec{X}$ that stores $N$ elements of $\mathbb{Z}/p\mathbb{Z}$, we assume that consecutive digits of one element are stored in adjacent machine-words in the memory. To this end, we view the input vector $\vec{X}$ as the row-major layout of a matrix $\mathtt{M_0}$ with $N$ rows and $k$ columns. We will refer to $\mathtt{M_0}$ as *non-transposed* input. Figures 4.1 and 4.2 show the way $\vec{X}$ is stored in $\mathtt{M_0}$.

$$\mathtt{M}_0 = \begin{bmatrix} \vec{X}_{(0,0)} & \vec{X}_{(0,1)} & \dots & \vec{X}_{(0,k-1)} \\ \vec{X}_{(1,0)} & \vec{X}_{(2,1)} & \dots & \vec{X}_{(1,k-1)} \\ \vdots & & & \\ \vdots & & & \\ \vec{X}_{(N-1,0)} & \vec{X}_{(N-1,1)} & \dots & \vec{X}_{(N-1,k-1)} \end{bmatrix}_{(N\times k)}$$

Figure 4.1: The non-transposed input matrix $\mathtt{M_0}$.

$$\mathtt{M}_0 = \begin{bmatrix} \mathtt{M}_0[0] & \mathtt{M}_0[1] & \dots & \mathtt{M}_0[k-1] \\ \mathtt{M}_0[k] & \mathtt{M}_0[k+1] & \dots & \mathtt{M}_0[2k-1] \\ \vdots & & & \\ \vdots & & & \\ \mathtt{M}_0[(N-1)*k] & \mathtt{M}_0[(N-1)*k+1] & \dots & \mathtt{M}_0[(N-1)*k+k-1] \end{bmatrix}_{(N\times k)}$$

Figure 4.2: Indexes of digits in the non-transposed matrix $\mathtt{M_0}$.

Assume that with $N$ threads running in parallel, a thread of index $\mathtt{tid}$ will have access to digits of $\vec{X}_{(\mathtt{tid})}$. Recall that the digit $\vec{X}_{(\mathtt{tid},i)}$ is stored at $\mathtt{M}[\mathtt{tid}*k+i]$ in the memory. A thread of index $\mathtt{tid}$ $(0 \leq \mathtt{tid} < N)$ will have access to the following memory addresses:

$$\vec{X}_{(\texttt{tid})} \mapsto (\texttt{M}[\texttt{tid} * \texttt{k}], \ldots, \texttt{M}[\texttt{tid} * \texttt{k} + (\texttt{k} - 1)]),$$
$$0 \leq \texttt{tid} < N.$$

As shown in Figure 4.3, all threads inside a warp will attempt to read $i$-th digit from their respective element, $\vec{X}_{\texttt{tid}}$, at the same time.

$$\begin{bmatrix} & \texttt{tid} = 0 & \texttt{tid} = 1 & \ldots & \texttt{tid} = 31 \\ & & & & \\ \texttt{i} = 0 & [\mathbf{0}, 1, \ldots, k-1] & [\mathbf{k}, k+1, \ldots, 2k-1] & \ldots & [\mathbf{31k}, 31k+1, \ldots, 32k-1] \\ \texttt{i} = 1 & [0, \mathbf{1}, \ldots, k-1] & [k, \mathbf{k+1}, \ldots, 2k-1] & \ldots & [31k, \mathbf{31k+1}, \ldots, 32k-1] \\ \vdots & \vdots & \vdots & & \vdots \\ \texttt{i} = \texttt{k} - 1 & [0, 1, \ldots, \mathbf{k\text{-}1}] & [k, k+1, \ldots, \mathbf{2k\text{-}1}] & \ldots & [31k, 31k+1, \ldots, \mathbf{32k\text{-}1}] \end{bmatrix}$$

Figure 4.3: Threads inside a warp reading from the non-transposed input.

In the context of CUDA programming, this way of handling memory is known as *strided access* pattern [7][1]. Strided accesses cause tremendous instruction overheads and are usually handled in the following way:

1. either by using shared memory (which, as we explained before, is not applicable), or
2. by transposing the input.

By the second solution, we transpose $\texttt{M}_0$ into a matrix $\texttt{M}_1$ with $k$ rows and $N$ columns. Figures 4.4 and 4.5 show how each element of $\vec{X}$ is stored in $\texttt{M}_1$.

$$\texttt{M}_1 = (\texttt{M}_0)^\top = \begin{bmatrix} X_{(0,0)} & X_{(1,0)} & \ldots & X_{(N-1,0)} \\ X_{(0,1)} & X_{(1,1)} & \ldots & X_{(N-1,1)} \\ \vdots & & & \\ X_{(0,k-1)} & X_{(1,k-1)} & \ldots & X_{(N-1,k-1)} \end{bmatrix}_{(k \times N)}$$

Figure 4.4: The transposed input matrix $\texttt{M}_1$.

[1]http://docs.nvidia.com/cuda/pdf/CUDA_C_Best_Practices_Guide.pdf

$$\mathtt{M}_1 = \begin{bmatrix} \mathtt{M}_1[0] & \mathtt{M}_1[1] & \dots & \mathtt{M}_1[N-1] \\ \mathtt{M}_1[N] & \mathtt{M}_1[N+1] & \dots & \mathtt{M}_1[2N-1] \\ \vdots & & & \\ \vdots & & & \\ \mathtt{M}_1[(k-1)*N] & \mathtt{M}_1[(k-1)*N+1] & \dots & \mathtt{M}_1[(k-1)*N+N-1] \end{bmatrix}_{(N\times k)}$$

Figure 4.5: Indexes of digits in the transposed matrix $\mathtt{M_1}$.

Therefore, digits of the element $\vec{X}_{(\mathtt{tid})}$ $(0 \leq \mathtt{tid} < N)$ are stored at the following memory addresses:

$$\vec{X}_{(\mathtt{tid})} \mapsto (\mathtt{M}[0*N+\mathtt{tid}], \mathtt{M}[1*N+\mathtt{tid}], \dots, \mathtt{M}[(k-1)*N+\mathtt{tid}]).$$

In other words, digits of the same index $i$ from all elements are stored in consecutive machine-words in memory. Therefore, each thread can have access to one digit of its respective element without lowering the memory efficiency.

As shown in Figure 4.6, threads inside a warp read the $i$-th digit from their respective element $\vec{X}_{\mathtt{tid}}$, therefore, all accesses to the memory will be in a coalesced way:

$$\begin{bmatrix} & \mathtt{tid}=0 & \mathtt{tid}=1 & \dots & \mathtt{tid}=31 \\ & & & & \\ at\ \mathtt{i}=0 & [\mathbf{0*N}] & [\mathbf{0*N+1}] & \dots & [\mathbf{0*N+31}] \\ at\ \mathtt{i}=1 & [\mathbf{1*N}] & [\mathbf{1*N+1}] & \dots & [\mathbf{1*N+31}] \\ \vdots & \vdots & \vdots & & \vdots \\ at\ \mathtt{i}=\mathtt{k} & [\mathbf{k*N}] & [\mathbf{k*N+1}] & \dots & [\mathbf{k*N+31}] \end{bmatrix}$$

Figure 4.6: Threads inside a warp reading from the transposed input.

The transposition of $M_0$ to $M_1$ can be computed on either the host (CPU) or the device (GPU). Notice that this transposition will not cause overheads, as it only needs to be computed once before transferring data to global memory, and once before writing the results back to the host memory.

Template $\mathsf{HostNaiveTranspose}(\vec{\mathtt{M}}, \mathtt{N}, \mathtt{k})$ gives a naive solution for computing transposition of elements of the input vector. In practice, such a computation is very inefficient as it has unoptimized accesses to the memory. In Chapter 5, we present algorithms for efficient transposition of data on GPUs.

**Template** HostNaiveTranspose($\vec{X}$, N, k)

**input:**

- two positive integers N and k, with $k$ as above,
- a vector $\vec{X}$ of $N \times k$ machines-words viewed as the row-major layout of a matrix $M_0$ with N rows and k columns.

**output:**

- a vector $\vec{X}$ storing the row-major layout of the transposed matrix of $M_0$.

**local:** vector $\vec{Y}$ viewed as the row-major layout of a matrix $M_1$ with N rows and k columns.

**for** $(0 \le i < N)$ **do**
  **for** $(0 \le j < L)$ **do**
    $\vec{Y}[j * N + i] := \vec{X}[i * L + j]$
  **end for**
**end for**
$\vec{X}[0 : k * N - 1] := \vec{Y}[0 : k * N - 1]$
**return** $\vec{X}$

## 4.2 Implementing big prime field arithmetic on GPUs

In this section, we explain the general structure of our kernels. Moreover, we present algorithms for computing arithmetic in $\mathbb{Z}/p\mathbb{Z}$ on GPUs.

### 4.2.1 Host entry point for arithmetic kernels

Assume that input vectors $\vec{X}$ and $\vec{Y}$, and the output vector $\vec{U}$, each store $N$ elements of $\mathbb{Z}/p\mathbb{Z}$. For computing a component-wise arithmetic operation, namely, operation (which can be replaced with any of the arithmetic operations in $\mathbb{Z}/p\mathbb{Z}$), for each of input vectors, one thread will be assigned for computing one element of the final result. Therefore, a thread of the index tid will compute the following element:

$$\vec{U}_{(\texttt{tid})} := \mathsf{operation}(\vec{X}_{(\texttt{tid})}, \vec{Y}_{(\texttt{tid})}). \tag{4.1}$$

Template HostGeneralOperation is a general example that presents the sequence of function calls for computing any of arithmetic operations in $\mathbb{Z}/p\mathbb{Z}$ on GPUs.

Initially, the host (CPU) function HostGeneralOperation invokes KernelGeneralOperation.

Then, function KernelGeneralOperation uses $N$ threads. Each thread of index `tid` will compute the following steps.

1. Each thread reads digits of $\vec{X}_{\mathtt{tid}}$ and $\vec{Y}_{\mathtt{tid}}$, then writes those digits into two vectors $\vec{x}$ and $\vec{y}$, respectively.
2. Then, each thread calls device function $\mathsf{operation}(\vec{x}, \vec{y})$.
3. In the next step, the invoked device function computes and returns the result, which will be stored in another vector $\vec{u}$.
4. Finally, kernel KernelGeneralOperation writes back the result $\vec{u}$ to $\vec{\mathtt{U}}_{\mathtt{tid}}$.

**Template** $\mathsf{HostGeneralOperation}(\vec{\mathtt{X}}, \vec{\mathtt{Y}}, \vec{\mathtt{U}}, \mathtt{N}, \mathtt{k}, \mathtt{r}, \mathtt{b})$

**Input:**

- a positive integer $b$ giving the size of a one dimensional thread block,
- two positive integers $k$ and $r$ as specified in the introduction,
- a positive integer $N$,
- two vectors $\vec{\mathtt{X}}$ and $\vec{\mathtt{Y}}$, each of them having $N$ elements of $\mathbb{Z}/p\mathbb{Z}$ with $p = r^k + 1$, thus each storing $\mathtt{N} \times \mathtt{k}$ machine-words.

**Output:**

- vector $\vec{\mathtt{U}}$ of elements in $\mathbb{Z}/p\mathbb{Z}$ storing the result $(\vec{\mathtt{U}} := \mathsf{operation}(\vec{\mathtt{X}}, \vec{\mathtt{Y}}))$.

$\vec{\mathtt{X}} := \mathsf{HostTranspose}(\vec{\mathtt{X}}, \mathtt{N}, \mathtt{k})$

$\vec{\mathtt{Y}} := \mathsf{HostTranspose}(\vec{\mathtt{Y}}, \mathtt{N}, \mathtt{k})$

$\mathsf{KernelGeneralOperation}<<< \mathtt{N}/\mathtt{b}, \mathtt{b} >>>(\vec{\mathtt{X}}, \vec{\mathtt{Y}}, \vec{\mathtt{U}}, \mathtt{N}, \mathtt{k}, \mathtt{r})$

**return** $\vec{\mathtt{U}}$

**Template** $\mathsf{KernelGeneralOperation}(\vec{\mathtt{X}}, \vec{\mathtt{Y}}, \vec{\mathtt{U}}, \mathtt{N}, \mathtt{k}, \mathtt{r})$

**input:**

- two positive integers $k$ and $r$ as specified in the introduction,
- a positive integer $\mathtt{N}$,
- two vectors $\vec{\mathtt{X}}$ and $\vec{\mathtt{Y}}$, each of them having $N$ elements of $\mathbb{Z}/p\mathbb{Z}$ with $p = r^k + 1$, thus each storing $\mathtt{N} \times \mathtt{k}$ machine-words and viewed as the row-major layout of the transposition of matrices $M_0$ and $M_1$ with $N$ rows and $k$ columns, respectively.

**output:**

- vector $\vec{\mathtt{U}}$ of elements in $\mathbb{Z}/p\mathbb{Z}$ storing the result ($\vec{\mathtt{U}} := \mathsf{operation}(\vec{\mathtt{X}}, \vec{\mathtt{Y}})$), viewed as the row-major layout of the transposition of a matrix $M_2$ with $N$ rows and $k$ columns.

**local:** $\mathtt{stride} := \mathtt{N}$

**local:** `offset:=0`

**local:** vectors $\vec{\mathtt{x}}$, $\vec{\mathtt{y}}$, $\vec{\mathtt{u}}$ each storing $\mathtt{k}$ digits of size of a machine-word, all digits initially set to 0.

**local:** `tid := blockIdx.x*blockSize.x+threadIdx.x`

**for** $(0 \leq \mathtt{i} < \mathtt{k})$ **do**

  `offset:=tid +i*stride`

  $\vec{\mathtt{x}}[\mathtt{i}] := \vec{\mathtt{X}}[\mathtt{offset}]$ ▷ Reading the digit with the index $\mathtt{i}$ of element $\vec{\mathtt{X}}_{\mathtt{tid}}$.

  $\vec{\mathtt{y}}[\mathtt{i}] := \vec{\mathtt{Y}}[\mathtt{offset}]$ ▷ Reading the digit with the index $\mathtt{i}$ of element $\vec{\mathtt{Y}}_{\mathtt{tid}}$.

**end for**

$\vec{\mathtt{u}} := \mathsf{DeviceGeneralOperation}(\vec{\mathtt{x}}, \vec{\mathtt{y}}, \mathtt{k}, \mathtt{r})$. ▷ each thread computing one element of the final result.

**for** $(0 \leq \mathtt{i} < \mathtt{k})$ **do**

  `offset:=tid +i*stride`

  $\vec{\mathtt{U}}[\mathtt{offset}] := \vec{\mathtt{u}}[\mathtt{i}]$

**end for**

**return** ▷ End of Kernel

**Template** DeviceGeneralOperation($\vec{\mathtt{x}}, \vec{\mathtt{y}}, \mathtt{k}, \mathtt{r}$)

**input:**
- two positive integers $k$ and $r$ as specified in the introduction,
- vectors $\vec{\mathtt{x}}$ and $\vec{\mathtt{y}}$ representing two elements of $\mathbb{Z}/p\mathbb{Z}$ with $p = r^k + 1$.

**output:**
- vector $\vec{\mathtt{u}}$ representing an element of $\mathbb{Z}/p\mathbb{Z}$ with $p = r^k + 1$.

**local:** vector $\vec{\mathtt{u}}$ storing $\mathtt{k}$ digits of size of a machine-word, all digits initially set to 0.
$\vec{\mathtt{u}} :=$ operation($\vec{\mathtt{x}}, \vec{\mathtt{y}}$) ▷ each thread computing one element of the final result.
**return** $\vec{\mathtt{u}}$

### 4.2.2 Implementation notes

In this section, we discuss two main parameters that can affect the performance of our CUDA implementation.

**Size of the thread block.** Recall that our aim is to maximize the memory efficiency. As we explained before, one thread will compute result of one arithmetic operation over $\mathbb{Z}/p\mathbb{Z}$, therefore, none of arithmetic operations depend on the size of a thread block. So, we must choose the size of a thread block by considering following metrics:

1. the achieved occupancy percentage,
2. the value of IPC (instruction per clock cycle), and
3. bandwidth-related performance metrics such as the load and store throughput.

Furthermore, we must limit the number of registers that can be allocated to each streaming multiprocessor. As we explain in Section 4.3, we have achieved the best experimental results for thread blocks of 128 threads and 256 threads.

**Chosen prime number.** Our current implementation is optimized for the prime $p = r^8 + 1$ with radix $r = 2^{63} + 2^{34}$. The radix $r$ is 63 bits wide, therefore we rely on 64-bit instructions on GPUs. As it is explained in [8], even though 64-bit integer arithmetic is supported on GPUs, at compile time, all arithmetic and memory instructions will first be converted to a sequence of 32-bit instructions. This might have a negative impact on the overall performance of our implementation. Specially, compared to addition and subtraction, 64-bit multiplication is computed through a longer sequence of 32-bit instructions.

### 4.2.3 Addition and subtraction in $\mathbb{Z}/p\mathbb{Z}$

In this section, we present algorithms for computing addition and subtraction in $\mathbb{Z}/p\mathbb{Z}$ based on the formulas in Chapter 3. Also, we assume that the input data is transposed in the way we explained in Section 4.1.4.

Algorithm 4.1 computes addition for two elements of $\mathbb{Z}/p\mathbb{Z}$. Using this algorithm with the higher level function KernelGeneralOperation, the component-wise addition for two vectors of $N$ elements of $\mathbb{Z}/p\mathbb{Z}$ can be computed in the following way:

$$\vec{\mathsf{U}} := \vec{\mathsf{X}} + \vec{\mathsf{Y}}.$$

Therefore, function KernelGeneralOperation goes through the following steps.

1. First, the algorithm reads the input data from $\vec{\mathsf{X}}$ and $\vec{\mathsf{Y}}$ write them to the local vectors $\vec{\mathsf{x}}$ and $\vec{\mathsf{y}}$.
2. Then, the algorithm passes $\vec{\mathsf{x}}$ and $\vec{\mathsf{y}}$ to device function DeviceAddition.
3. Finally, the algorithm writes back the result $\vec{\mathsf{u}}$ to the transposed output vector $\vec{\mathsf{U}}$.

In a similar way, Algorithm 4.2 computes subtraction for two elements of $\mathbb{Z}/p\mathbb{Z}$. Using this algorithm with the higher level function KernelGeneralOperation, the component-wise subtraction for two vectors of $N$ elements of $\mathbb{Z}/p\mathbb{Z}$ can be computed in the following way:

$$\vec{\mathsf{U}} := \vec{\mathsf{X}} - \vec{\mathsf{Y}}.$$

**Algorithm 4.1** DeviceAddition($\vec{\mathtt{x}}, \vec{\mathtt{y}}, \mathtt{k}, \mathtt{r}$)

**input:**

- two positive integers $k$ and $r$ as specified in the introduction,
- vectors $\vec{\mathtt{x}}$ and $\vec{\mathtt{y}}$ representing two elements of $\mathbb{Z}/p\mathbb{Z}$ with $p = r^k + 1$.

**output:**

- vector $\vec{\mathtt{u}}$ representing an element of $\mathbb{Z}/p\mathbb{Z}$ with $p = r^k + 1$, storing result of the addition ($\vec{\mathtt{u}} := \vec{\mathtt{x}} + \vec{\mathtt{y}}$) in $\mathtt{k}$ digits of size of a machine-word.

**local:** $\mathtt{c} := 0,\ \mathtt{sum} := 0$

**local:** vector $\vec{\mathtt{u}}$ storing $\mathtt{k}$ digits of size of a machine-word, set all digits of $\vec{\mathtt{u}}$ equal to zero.

```
for (0 ≤ i ≤ k − 1) do
    sum := (x⃗[i] + y⃗[i] + c);                         ▷ 0 ≤ sum < 2r
    if sum < x⃗[i] or sum < y⃗[i] then                  ▷ An overflow has happened here.
        c := 1;                                       ▷ The carry flag will be set to 1.
    else if sum ≥ r then        ▷ There is no overflow but sum is greater than radix.
        c := 1;   ▷ The carry flag will be set to 1, adding 1 to u⃗[i + 1] in the next step.
        sum := sum − r;
    end if
    u⃗[i] := sum
end for

if c = 1 then                 ▷ The sum is greater than r^k, so add r^k = −1 mod p.
    j := −1
    Find the index j where u⃗[j] is the first non-zero integer in u⃗
    if j ≠ −1 then                  ▷ This means u⃗[0], u⃗[1], . . . , u⃗[j − 1] are zero.
        u⃗[j] := u⃗[j] − 1;                       ▷ the lower borrows r from higher.
        for (0 ≤ i ≤ j − 1) do
            u⃗[i] := r − 1;
        end for
    else                        ▷ j = −1 which means all elements in u⃗ are zero.
        u⃗[0] := 2^64 − 1;                          ▷ Therefore, set u⃗ := −1 mod p.
        u⃗[1], . . . , u⃗[k − 1] := 0;
    end if
end if
return u⃗
```

**Algorithm 4.2** DeviceSubtraction($\vec{x}, \vec{y}, k, r$)

**input:**

- two positive integers $k$ and $r$ as specified in the introduction,
- vectors $\vec{x}$ and $\vec{y}$ representing two elements of $\mathbb{Z}/p\mathbb{Z}$ with $p = r^k + 1$.

**output:**

- vector $\vec{u}$ representing an element of $\mathbb{Z}/p\mathbb{Z}$ with $p = r^k + 1$, storing result of the addition ($\vec{u} := \vec{x} - \vec{y}$) in k digits of size of a machine-word.

**local:** c := 0, s := 0

**local:** vector $\vec{u}$ storing k digits of size of a machine-word, all digits initially set to 0.

```
for (0 ≤ i ≤ s − 1) do
    s := (ỹ[i] + c);                                   ▷ 0 ≤ s ≤ r
    if s < x⃗[i] then                                   ▷ x⃗[i] need to borrow r from u⃗[i + 1].
        u⃗[i] := s + r − x⃗[i]
        c := 1;                                        ▷ The carry flag will be set to 1.
    else
        u⃗[i] := x⃗[i] − s;
    end if
end for
if c = 1 then                                          ▷ The value of u is less than x − y, then add r^k to u.
    j := −1
    Find the index j where u⃗[j] is first digit in u⃗ smaller than r − 1.
    if j ≠ −1 then                                     ▷ This means u⃗[0], u⃗[1], ..., u⃗[j − 1] are equal to r − 1.
        u⃗[j] := u⃗[j] + 1;
        for (0 ≤ i ≤ j − 1) do
            u⃗[i] := 0;
        end for
    else                                               ▷ j = −1 which means all digits in u⃗ are zero.
        u⃗[0] := 2^64 − 1;                              ▷ Therefore, set u := −1 mod p.
        u⃗[1], ..., u⃗[k − 1] := 0;
    end if
end if
return u⃗
```

### 4.2.4 Multiplication by a power of r in $\mathbb{Z}/p\mathbb{Z}$

In this section, we present algorithms for computing multiplication by powers of radix in $\mathbb{Z}/p\mathbb{Z}$. As we explained in Section 3.4, a multiplication by a power of radix can be reduced to a *rotation* followed by one subtraction over $\mathbb{Z}/p\mathbb{Z}$. This multiplication requires $\mathcal{O}(k)$ machine-word operations.

**Vector rotation.** Rotation is a simple *data movement* primitive. This operation is used as a part of the multiplication by powers of radix. As input, the algorithm takes the local array $\vec{x}$, which stores $k$ digits of size of a machine-word, then, simply moves all elements of $\vec{x}$ one unit to the right. Algorithm 4.3 presents a pseudo-code for this operation.

**Algorithm 4.3** DeviceRotation($\vec{\mathtt{x}}, \mathtt{k}$)

**input:**
- a positive integers $k$ as specified in the introduction,
- vector $\vec{\mathtt{x}}$ storing $\mathtt{k}$ digits of size of a machine-word.

**output:**
- vector $\vec{\mathtt{x}}$ storing the result in $\mathtt{k}$ digits of size of a machine-word.

**local:** $\mathtt{t} := \vec{\mathtt{x}}[\mathtt{k}-1]$
**for** ($\mathtt{i}$ from ($\mathtt{k}-1$) to 1 by $-1$) **do**
  $\vec{\mathtt{x}}[\mathtt{i}] := \vec{\mathtt{x}}[\mathtt{i}-1]$ ▷ Each digit of the $\vec{\mathtt{x}}$ is moved one unit to the right.
**end for**
$\vec{\mathtt{x}}[0] := \mathtt{t}$
**return** $\vec{\mathtt{x}}$

Algorithm 4.4 presents a solution for computing multiplication of one element of $\mathbb{Z}/p\mathbb{Z}$ by a power of $r$.

At a lower level, function DeviceMultPowR computes $\vec{x} * r^s$ in the following steps.

1. First, the algorithm allocates two local vectors $\vec{a}$ and $\vec{b}$, each of them storing $k$ digits of size of a machine-word, all digits initially set to 0.
2. Then, the algorithm proceeds by storing the higher $k-s$ digits of $\vec{x}$ into $\vec{\mathtt{a}}$.
3. In the next step, the algorithm stores the lower $s$ digits of $\vec{x}$ into $\vec{\mathtt{b}}$.
4. The algorithm continues by computing DeviceRotation($\vec{\mathtt{b}}$), $s$ times in a row.
5. After that, the algorithm negates $\vec{\mathtt{a}}$ by computing $\vec{\mathtt{a}} := \vec{0} - \vec{\mathtt{a}}$.
6. Finally, the algorithm computes $\vec{\mathtt{u}} := \vec{\mathtt{a}} + \vec{\mathtt{b}}$, then, returns the vector $\vec{\mathtt{u}}$.

**Algorithm 4.4** $\mathsf{DeviceMultPowR}(\vec{\mathtt{x}}, \mathtt{s}, \mathtt{k}, \mathtt{r})$

**input:**

- two positive integers $k$ and $r$ as specified in the introduction,
- a positive integer $s$ representing power of radix $r^s$ $(\mathtt{0} < \mathtt{s} \leq \mathtt{k})$,
- vector $\vec{\mathtt{x}}$ representing one elements of $\mathbb{Z}/p\mathbb{Z}$ with $p = r^k + 1$.

**output:**

- vector $\vec{\mathtt{x}}$ stores result $(\vec{\mathtt{x}} := \vec{\mathtt{x}} * r^s)$ in $\mathtt{k}$ digits of size of a machine-word.

**local:** vectors $\vec{\mathtt{a}}$, $\vec{\mathtt{b}}$, $\vec{\mathtt{Z}}$ each store $\mathtt{k}$ digits of size of a machine-word, all digits initially set to 0.

$\vec{\mathtt{a}}[\mathtt{k} - \mathtt{s} : \mathtt{s} - \mathtt{1}] := \vec{\mathtt{x}}[\mathtt{k} - \mathtt{s} : \mathtt{s} - \mathtt{1}]$ ▷ Storing upper $\mathtt{s}$ digits of $\vec{\mathtt{x}}$ in $\vec{\mathtt{a}}$.

$\vec{\mathtt{b}}[\mathtt{0} : \mathtt{k} - \mathtt{s} - \mathtt{1}] := \vec{\mathtt{x}}[\mathtt{0} : \mathtt{k} - \mathtt{s} - \mathtt{1}]$ ▷ Storing lower $\mathtt{k} - \mathtt{s}$ digits of $\vec{\mathtt{x}}$ in $\vec{\mathtt{b}}$.

**for** $(\mathtt{0} \leq \mathtt{i} < \mathtt{s})$ **do**

  $\vec{\mathtt{b}} := \mathsf{DeviceRotation}(\vec{\mathtt{b}}, \mathtt{k})$ ▷ Shifting elements of $\vec{\mathtt{b}}$ one unit to the right.

**end for**

$\vec{\mathtt{a}} := \mathsf{DeviceSubtraction}(\vec{\mathtt{z}}, \vec{\mathtt{a}}, \mathtt{k}, \mathtt{r})$ ▷ Negating $\vec{\mathtt{a}}$ by computing $\vec{\mathtt{a}} := \vec{0} - \vec{\mathtt{a}}$.

$\vec{\mathtt{x}} := \mathsf{DeviceAddition}(\vec{\mathtt{a}}, \vec{\mathtt{b}}, \mathtt{k}, \mathtt{r})$ ▷ $\vec{\mathtt{x}} := \vec{\mathtt{a}} + \vec{\mathtt{b}}$.

**return** $\vec{\mathtt{x}}$

### 4.2.5 Multiplication in $\mathbb{Z}/p\mathbb{Z}$

In this section, we explain algorithms for computing component-wise multiplication of two vectors of $N$ elements of $\mathbb{Z}/p\mathbb{Z}$. This multiplication requires $\mathcal{O}(k^2)$ machine-word operations. For example, $\vec{U}_j := \vec{X}_j * \vec{Y}_j$ computes component-wise multiplication for $j$-th elements of $\vec{X}$ and $\vec{Y}$, respectively.

This product is computed similar to polynomial multiplication for two polynomials of degree $k$. However, it has a few additional steps. Currently, we have implemented plain multiplication algorithm as the default function for computing multiplications in $\mathbb{Z}/p\mathbb{Z}$.

Assume that two elements $X_j$ and $Y_j$ are indexed in the following way:

$$\vec{X}_j = (x_0, x_1, \ldots, x_{(k-1)}), \vec{Y}_j = (y_0, y_1, \ldots, y_{(k-1)}). \tag{4.2}$$

In the first step, the multiplication algorithm computes the *intermediate products* of the form $x_i y_j r^{(i+j)}$, then, adds them together to calculate the *intermediate results*. Then, the algorithm computes $2k$ *intermediate results* of the form $(l_m, h_m, c_m)$ for $0 \leq m < 2k$ in the following way:

$$
\begin{array}{cccclcc}
(l_{2k-1}, & h_{2k-1}, & c_{2k-1}) & = & 0 & \times & r^{2k-1} \\
(l_{2k-2}, & h_{2k-2}, & c_{2k-2}) & = & (x_{k-1}y_{k-1}) & \times & r^{2k-2} \\
(l_{2k-3}, & h_{2k-3}, & c_{2k-3}) & = & (x_{k-2}y_{k-1} + x_{k-1}y_{k-2}) & \times & r^{2k-3} \\
\vdots & \vdots & \vdots & & \vdots & \vdots & \\
(l_k, & h_k, & c_k) & = & (x_1y_{k-1} + \ldots + x_{k-2}y_1 + x_{k-1}y_0) & \times & r^k \\
(l_{k-1}, & h_{k-1}, & c_{k-1}) & = & (x_0y_{k-1} + x_1y_{k-2} + \ldots + x_{k-1}y_0) & \times & r^{k-1} \\
(l_{k-2}, & h_{k-2}, & c_{k-2}) & = & (x_0y_{k-2} + x_1y_{k-3} + \ldots + x_{k-2}y_0) & \times & r^{k-2} \\
(l_{k-3}, & h_{k-3}, & c_{k-3}) & = & (x_0y_{k-3} + x_1y_{k-4} + \ldots + x_{k-3}y_0) & \times & r^{k-3} \\
\vdots & \vdots & \vdots & & \vdots & \vdots & \\
(l_0, & h_0, & c_0) & = & (x_0y_0) & \times & r^0
\end{array}
\tag{4.3}
$$

By partitioning intermediate results of power $r^j$ and $r^{(j+k)}$ together we have:

$$
\begin{array}{cccclcl}
(l_{k-1}, & h_{k-1}, & c_{k-1}) & = & (x_0y_{k-1} + \ldots + x_{k-1}y_0)r^{k-1} & + & 0.r^{2k-1} \\
(l_{k-2}, & h_{k-2}, & c_{k-2}) & = & (x_0y_{k-2} + \ldots + x_{k-2}y_0)r^{k-2} & + & (x_{k-1}y_{k-1})r^{2k-2} \\
(l_{k-3}, & h_{k-3}, & c_{k-3}) & = & (x_0y_{k-3} + \ldots + x_{k-3}y_0)r^{k-3} & + & (x_{k-2}y_{k-1} + x_{k-1}y_{k-2})r^{2k-3} \\
\vdots & & \vdots & & \vdots & & \\
(l_0, & h_0, & c_0) & = & (x_0y_0)r^0 & + & (x_1y_{k-1} + \ldots + x_{k-2}y_1 + x_{k-1}y_0)r^k
\end{array}
$$

By this arrangement, the triple $[l_{(k-m-1)}, h_{(k-m-1)}, c_{(k-m-1)}]$ can be computed as follows.

1. First, all digits of $\vec{X}_j$ and $\vec{Y}_j$ will be stored in vectors $\vec{x}$ and $\vec{y}$, respectively.
2. Then, multiplication $\vec{y} := \vec{y}r^m$ will be computed by $\mathsf{DeviceMultPowR}(\vec{y}, m, k, r)$.
3. Also, for $0 \leq i < k$, $k$ independent products will be computed and stored in a vector of size of $3k$ machine-words, namely, $\vec{\mathtt{M}}$ in the following way:

$$\vec{\mathtt{M}}[3i, 3i+1, 3i+2] \;:=\; \mathsf{DeviceIntermediateProduct}(\vec{x}_i * \vec{y}_{(k-i)}, k, r). \tag{4.4}$$

4. Finally, the following sum will be computed, initially $[\mathtt{l}, \mathtt{h}, \mathtt{c}] := [0, 0, 0]$:

$$[l_{(k-m-1)}, h_{(k-m-1)}, c_{(k-m-1)}] := \sum_{i=0}^{k} \mathsf{DeviceAddition}([\mathtt{l}, \mathtt{h}, \mathtt{c}], \vec{\mathtt{M}}[3i, 3i+1, 3i+2], k, r). \tag{4.5}$$

This scheme decomposes computation of intermediate results into $k$ independent units of work. This implies a possibility for parallelizing this computation by using at most $k$ threads.

Vectors $\vec{L}, \vec{H}$, and $\vec{C}$ are used for storing $N$ elements of $\mathbb{Z}/p\mathbb{Z}$. These vectors store intermediate results for multiplication $\vec{X}_i * \vec{Y}_i$ in $\vec{L}_i$, $\vec{H}_i$, and $\vec{C}_i$, respectively. Finally, Algorithm 4.5 computes the final result of multiplication from intermediate results in the following way:

$$\vec{U}_i := \vec{L}_i + (\vec{H}_i)r + (\vec{C}_i)r^2. \tag{4.6}$$

**Algorithm 4.5** DeviceMultFinalResult($\vec{\mathtt{l}}, \vec{\mathtt{h}}, \vec{\mathtt{c}}, \mathtt{k}, \mathtt{r}$)

**input:**

- vectors $\vec{\mathtt{l}}, \vec{\mathtt{h}}, \vec{\mathtt{c}}$ representing three elements of $\mathbb{Z}/p\mathbb{Z}$ with $p = r^k + 1$, each storing intermediate results in $\mathtt{k}$ digits of size of a machine-word.

**output:**

- vector $\vec{\mathtt{t}}$ representing an element of $\mathbb{Z}/p\mathbb{Z}$ with $p = r^k + 1$, storing result of intermediate addition ($\vec{\mathtt{t}} := \vec{\mathtt{l}} + \vec{\mathtt{h}}r + \vec{\mathtt{c}}r^2$) in $\mathtt{k}$ digits of size of a machine-word.

**local:** vector $\vec{\mathtt{t}}$ storing temporary results in $\mathtt{k}$ digits of size of a machine-word, all digits initially set to 0.

$\vec{\mathtt{h}} :=$ DeviceMultPowR($\vec{\mathtt{h}}, 1, \mathtt{k}, \mathtt{r}$) $\triangleright\ \vec{\mathtt{h}} := \vec{\mathtt{h}}r$

$\vec{\mathtt{c}} :=$ DeviceMultPowR($\vec{\mathtt{c}}, 2, \mathtt{k}, \mathtt{r}$) $\triangleright\ \vec{\mathtt{c}} := \vec{\mathtt{c}}r^2$

$\vec{\mathtt{t}} :=$ DeviceAddition($\vec{\mathtt{l}}, \vec{\mathtt{h}}, \mathtt{k}, \mathtt{r}$) $\triangleright\ \vec{\mathtt{t}} := \vec{\mathtt{l}} + \vec{\mathtt{h}}r$

$\vec{\mathtt{t}} :=$ DeviceAddition($\vec{\mathtt{c}}, \vec{\mathtt{t}}, \mathtt{k}, \mathtt{r}$) $\triangleright\ \vec{\mathtt{t}} := \vec{\mathtt{l}} + \vec{\mathtt{h}}r + \vec{\mathtt{c}}r^2$

**return** $\vec{\mathtt{t}}$

Template DeviceIntermediateProduct computes intermediate products of two digits. A specific case of this template is presented in Algorithm 4.6, which computes products in $\mathbb{Z}/p\mathbb{Z}$, with $p = r^8 + 1$, and $r = 2^{63} + 2^{34}$.

In the rest of this section, we will explain Algorithms 4.7 and 4.9, which compute intermediate results in sequential and parallel ways, respectively.

**Template** DeviceIntermediateProduct($[\mathtt{a}, \mathtt{b}], \mathtt{k}, \mathtt{r}$)

**input:**

- two positive integers $k$ and $r$ as specified in the introduction,
- two digits $a$ and $b$ each of size of of a machine-word.

**output:**

- three positive integers $l, h$, and $c$ storing result of intermediate product.

$[\mathtt{l}, \mathtt{h}, \mathtt{c}] := (\mathtt{a} * \mathtt{b}) \mod (\mathtt{p})$

**return** $[l, h, c]$

**Algorithm 4.6** $\mathsf{DeviceIntermediateProduct1}([\mathtt{a},\mathtt{b}],\mathtt{k}:=8,\mathtt{r}:=2^{63}+2^{34})$

$\mathtt{a}:=0,\mathtt{b}:=0$
**local:** $\mathtt{x}_0,\mathtt{x}_1,\mathtt{y}_0,\mathtt{y}_1$
**local:** $\mathtt{x}_1=\mathtt{a}>=\mathtt{r}?1:0$
**local:** $\mathtt{x}_0=\mathtt{x}_1>0?\mathtt{a}-\mathtt{r}:\mathtt{x}$ ▷ $\mathtt{x}=\mathtt{x}_0+\mathtt{x}_1\mathtt{r}$
**local:** $\mathtt{y}_1=\mathtt{y}>=\mathtt{r}?1:0$
**local:** $\mathtt{y}_0=\mathtt{y}_1>0?\mathtt{y}-\mathtt{r}:\mathtt{y}$ ▷ $\mathtt{y}=\mathtt{y}_0+\mathtt{y}_1\mathtt{r}$
**local:** $[\mathtt{v}_1,\mathtt{v}_2,\mathtt{v}_3]=[0,\mathtt{x}_0\mathtt{y}_1,0]$ ▷ $\mathtt{x}_0\mathtt{y}_1\mathtt{r}$
**local:** $[\mathtt{v}_4,\mathtt{v}_5,\mathtt{v}_6]=[0,\mathtt{x}_1\mathtt{y}_0,0]$ ▷ $\mathtt{x}_1\mathtt{y}_0\mathtt{r}$
**local:** $[\mathtt{v}_7,\mathtt{v}_8,\mathtt{v}_9]=[0,0,\mathtt{x}_1\mathtt{y}_1]$ ▷ $\mathtt{x}_1\mathtt{y}_1\mathtt{r}^2$
**local:** $[\mathtt{c}_0,\mathtt{c}_1]=\mathtt{func}(\mathtt{x}_0\mathtt{y}_0)$ ▷ $\mathtt{x}_0\mathtt{y}_0=\mathtt{c}_0+\mathtt{c}_1 2^{64}$
**local:** $[\mathtt{v}_{10},\mathtt{v}_{11},\mathtt{v}_{12}]=\mathtt{func2}(\mathtt{c}_0)$ ▷ $\mathtt{c}_0=\mathtt{v}_{10}+\mathtt{v}_{11}\mathtt{r}+\mathtt{v}_{12}\mathtt{r}^2$
**local:** $[\mathtt{v}_{13},\mathtt{v}_{14},\mathtt{v}_{15}]=\mathtt{func2}(\mathtt{c}_1'\mathtt{r})$ ▷ $\mathtt{c}_1'\mathtt{r}=\mathtt{v}_{13}+\mathtt{v}_{14}\mathtt{r}+\mathtt{v}_{15}\mathtt{r}^2;$
**local:** $\mathtt{d}_1=\mathtt{c}_1'>>29$
**local:** $\mathtt{d}_0=\mathtt{c}_1'-\mathtt{d}_1<<29$
**local:** $\mathtt{e}_1=(\mathtt{d}_0-\mathtt{d}_1)>>29$
**local:** $\mathtt{e}_0=(\mathtt{d}_0-\mathtt{d}_1-\mathtt{e}_1<<29)$
**local:** $[\mathtt{v}_{16},\mathtt{v}_{17},\mathtt{v}_{18}]=[(\mathtt{e}_0-\mathtt{e}_1)<<34,\mathtt{e}_1+\mathtt{d}_1,0]$
**local:** $[\mathtt{l},\mathtt{h},\mathtt{c}]=[\mathtt{v}_1+\mathtt{v}_4+\cdots+\mathtt{v}_{16},\mathtt{v}_2+\mathtt{v}_5+\cdots+\mathtt{v}_{17},\mathtt{v}_3+\mathtt{v}_6+\cdots+\mathtt{v}_{18}]$
**return** $[\mathtt{l},\mathtt{h},\mathtt{c}]$

### Sequential plain multiplication

In this section, we explain how we can compute intermediate results of multiplication in $\mathbb{Z}/p\mathbb{Z}$ in a sequential way.

Similar to addition, subtraction, and multiplication by powers of radix, one thread will be assigned for computing each element of the final result. Therefore, each thread will compute $k$ triples of the form $[l_{(k-m-1)}, l_{(k-m-1)}, c_{(k-m-1)}]$ for $0 \leq m < k$. Therefore, we assign $N$ threads for component-wise multiplication on two input vectors $\vec{X}$ and $\vec{Y}$ of size $N$. Every thread of index `tid` computes multiplication in the following steps.

**Step I.** First, each thread reads digits of elements $\vec{X}_{\mathtt{tid}}$ and $\vec{Y}_{\mathtt{tid}}$ to the vectors $\vec{x}$ and $\vec{y}$ in the following way:

$$\begin{aligned} \vec{x}[0:k-1] &:= (\vec{X}_{(\mathtt{tid},0)},\ldots,\vec{X}_{(\mathtt{tid},k-1)}),\\ \vec{y}[0:k-1] &:= (\vec{Y}_{(\mathtt{tid},0)},\ldots,\vec{Y}_{(\mathtt{tid},k-1)}). \end{aligned} \tag{4.7}$$

**Step II.** Each thread computes $k$ iterations, when at each iteration $i$ $(0 \leq i < k)$ the thread goes through the following steps.

1. First, the thread computes $\vec{Y}_{\mathtt{tid}} * r^1$.
2. Next, the thread proceeds with computing $[l, h, c] = \sum_{0 \leq m < k} (\vec{x}_m * \vec{y}_{k-m})$.
3. Lastly, the thread stores the values of the triple $[l, h, c]$ to the corresponding addresses in global memory:

$$\vec{L}_{(\mathtt{tid},k-i-1)} = l, \vec{H}_{(\mathtt{tid},k-i-1)} = h, \vec{C}_{(\mathtt{tid},k-i-1)} = c. \tag{4.8}$$

**Step III.** Finally, each thread computes the final result in the following way:

$$\vec{U}_{\mathtt{tid}} = \vec{L}_{(\mathtt{tid})} + \vec{H}_{(\mathtt{tid})} r + \vec{C}_{(\mathtt{tid})} r^2. \tag{4.9}$$

Algorithm 4.7 presents a sequential solution for computing intermediate results. This algorithm depends on Algorithm 4.8 for computing $k$ iterations of Step II.

**Algorithm 4.7** KernelSequentialPlainMult($\vec{\mathtt{X}}, \vec{\mathtt{Y}}, \vec{\mathtt{U}}, \mathtt{N}, \mathtt{k}, \mathtt{r}$)

**input:**

- two positive integers $k$ and $r$ as specified in the introduction,
- a positive integer $\mathtt{N}$,
- two vectors $\vec{\mathtt{X}}$ and $\vec{\mathtt{Y}}$, each of them having $N$ elements of $\mathbb{Z}/p\mathbb{Z}$ with $p = r^k + 1$, thus each storing $\mathtt{N} \times \mathtt{k}$ machine-words, viewed as the row-major layout of the transposition of two matrices $M_0$ and $M_1$ with $N$ rows and $k$ columns, respectively.

**output:**

- vector $\vec{\mathtt{U}}$ of elements in $\mathbb{Z}/p\mathbb{Z}$ storing the the result ($\vec{\mathtt{U}} := \vec{\mathtt{X}} * \vec{\mathtt{Y}}$), viewed as the row-major layout of the transposition of a matrix $M_2$ with $N$ rows and $k$ columns.

**local:** `offset:=0`

**local:** `tid := blockIdx.x*blockSize.x+threadIdx.x`

**local:** vectors $\vec{\mathtt{x}}$, $\vec{\mathtt{y}}$, $\vec{\mathtt{u}}$, $\vec{\mathtt{l}}$, $\vec{\mathtt{h}}$, $\vec{\mathtt{c}}$ each storing $\mathtt{k}$ digits of size of a machine-word, all digits initially set to 0.

**for** $(0 \leq \mathtt{i} < \mathtt{k})$ **do**
  $\mathtt{offset} := \mathtt{tid} + \mathtt{i} * \mathtt{N}$
  $\vec{\mathtt{x}}[\mathtt{i}] := \vec{\mathtt{X}}[\mathtt{offset}]$
  $\vec{\mathtt{y}}[\mathtt{i}] := \vec{\mathtt{Y}}[\mathtt{offset}]$
**end for**
$[\vec{\mathtt{l}}, \vec{\mathtt{h}}, \vec{\mathtt{c}}] :=$ DeviceSequentialMult($\vec{\mathtt{x}}, \vec{\mathtt{y}}, \mathtt{k}, \mathtt{r}$) ▷ each thread computing $k$ digits.
$\vec{\mathtt{u}} :=$ DeviecMultFinalResult($\vec{\mathtt{l}}, \vec{\mathtt{h}}, \vec{\mathtt{c}}, \mathtt{k}, \mathtt{r}$)
**for** $(0 \leq \mathtt{i} < \mathtt{k})$ **do**
  $\mathtt{offset} := \mathtt{tid} + \mathtt{i} * \mathtt{N}$
  $\vec{\mathtt{U}}[\mathtt{offset}] := \vec{\mathtt{u}}[\mathtt{i}]$
**end for**
**return** ▷ End of Kernel

**Algorithm 4.8** DeviceSequentialMult($\vec{\mathtt{x}}, \vec{\mathtt{y}}, \mathtt{k}, \mathtt{r}$)

**input:**

- two positive integers $k$ and $r$ as specified in the introduction,
- vectors $\vec{\mathtt{x}}$ and $\vec{\mathtt{y}}$ representing two elements of $\mathbb{Z}/p\mathbb{Z}$ with $p = r^k + 1$.

**output:**

- vectors $\vec{\mathtt{l}}, \vec{\mathtt{h}}, \vec{\mathtt{c}}$ representing three elements of $\mathbb{Z}/p\mathbb{Z}$ with $p = r^k + 1$, each storing intermediate results in $\mathtt{k}$ digits of size of a machine-word.

**local:** vectors $\vec{\mathtt{s}}, \vec{\mathtt{t}}$ each storing the result of intermediate additions in $\mathtt{k}$ digits of size of a machine-word.

**local:** vectors $\vec{\mathtt{l}}, \vec{\mathtt{h}}, \vec{\mathtt{c}}$ storing result in $\mathtt{k}$ digits of size of a machine-word.

```
for (0 ≤ i < k) do
    if i > 0 then
        y⃗ := DeviceMultPowR(y⃗, 1)                          ▷ y⃗ := y⃗r
    end if
    set s⃗ := [0, 0, 0] and t⃗ := [0, 0, 0]
    for (0 ≤ j < k) do
        t⃗ := DeviceIntermediateProduct(x⃗[j], y⃗[k − j], k, r)
        s⃗ := DeviceAddition(s⃗, t⃗, k, r)        ▷ computing addition for 3 digits.
    end for
    l⃗[k − i − 1] := s⃗[0]   ▷ storing lower part of intermediate result in the output vector l⃗.
    h⃗[k − i − 1] := s⃗[1]   ▷ storing higher part of intermediate result in the output vector h⃗.
    c⃗[k − i − 1] := s⃗[2]   ▷ storing carry part of intermediate result in the output vector c⃗.
end for
return [l⃗, h⃗, c⃗]
```

**Parallel plain multiplication using $k$ threads**

As we explained before, intermediate results can be computed in a parallel way. For this purpose, $n$ threads can be utilized ($2 \leq n \leq k$) for computing $k$ triples like $[l_{(k-m-1)}, l_{(k-m-1)}, c_{(k-m-1)}]$, with $0 \leq m < k$.

We explain the case of $n = k$, where each thread computes one triple of intermediate results. Consequently, for input vectors $\vec{X}, \vec{Y}$ of size $N$, algorithm assigns $k \times N$ threads.

Then, a thread of index `tid` goes through the following steps.

**Step I.**

1. First, the thread reads all $k$ digits from element $\vec{X}_{\mathtt{tid}/k}$ to a local vector $\vec{x}$ such that

$$\begin{aligned} \vec{x}[0:k-1] &:= (\vec{X}_{(\mathtt{tid}/k,0)}, \ldots, X_{(\mathtt{tid}/k,k-1)}), \\ \vec{y}[0:k-1] &:= (\vec{Y}_{(\mathtt{tid}/k,0)}, \ldots, \vec{Y}_{(\mathtt{tid}/k,k-1)}). \end{aligned} \tag{4.10}$$

2. The thread computes the index of its relative digit $i := (\mathtt{tid} \mod k)$.
3. The thread computes multiplication $\vec{y} * r^i$.
4. The thread computes $k$ intermediate products and adds them together:

$$[l, h, c] := \sum_{0 \leq m < k} (\vec{x}_m * \vec{y}_{k-m}). \tag{4.11}$$

5. The thread writes intermediate results to vectors $\vec{L}, \vec{H}, \vec{C}$ such that

$$L_{(\mathtt{tid}/k,k-i-1)} := l, H_{(\mathtt{tid}/k,k-i-1)} := h, C_{(\mathtt{tid}/k,k-i-1)} = c. \tag{4.12}$$

**Step II.** At this point, all intermediate results are stored in vectors $\vec{L}, \vec{H}, \vec{C}$, therefore the algorithm uses $N$ threads, with each thread computing on element of the final result by using Algorithm 4.5. At the end

$$\vec{U} = \vec{L}_{\mathtt{tid}} + (\vec{H_{\mathtt{tid}}})r + (\vec{C}_{\mathtt{tid}})r^2. \tag{4.13}$$

Algorithm 4.9 presents a parallel solution for computing intermediate results of multiplication in $\mathbb{Z}/p\mathbb{Z}$. At its core, this algorithm depends on Algorithm 4.10 for computing Step I.

**Algorithm 4.9** KernelParallelPlainMult($\vec{\mathtt{X}}, \vec{\mathtt{Y}}, \vec{\mathtt{U}}, \vec{\mathtt{L}}, \vec{\mathtt{H}}, \vec{\mathtt{C}}, \mathtt{N}, \mathtt{k}, \mathtt{r}$)

**input:**

- two positive integers $k$ and $r$ as specified in the introduction,
- a positive integer $\mathtt{N}$,
- vectors $\vec{\mathtt{X}}$, $\vec{\mathtt{Y}}$, $\vec{\mathtt{L}}$, $\vec{\mathtt{H}}$, and $\vec{\mathtt{C}}$ each having $N$ elements of $\mathbb{Z}/p\mathbb{Z}$ with $p = r^k+1$, thus each storing $\mathtt{N} \times \mathtt{k}$ machine-words, viewed as the row-major layout of the transposition of matrices $M_0$, $M_1$, $M_2$, $M_3$, and $M_4$ respectively.

**output:**

- vector $\vec{\mathtt{U}}$ of elements in $\mathbb{Z}/p\mathbb{Z}$ storing the the result ($\vec{\mathtt{U}} := \vec{\mathtt{X}} * \vec{\mathtt{Y}}$), viewed as the row-major layout of the transposition of a matrix $M_5$ with $N$ rows and $k$ columns.

**local:** $\mathtt{offset} := 0$

**local:** $\mathtt{tid\ := blockIdx.x{*}blockSize.x{+}threadIdx.x}$

**local:** vectors $\vec{\mathtt{x}}$, $\vec{\mathtt{y}}$, $\vec{\mathtt{u}}$, $\vec{\mathtt{l}}$, $\vec{\mathtt{h}}$, and $\vec{\mathtt{c}}$, each storing $\mathtt{k}$ digits of size of a machine-word, all digits initially set to 0.

```
for (0 ≤ i < k) do
    offset := tid/k + i * N
    x⃗[i] := X⃗[offset]
    y⃗[i] := Y⃗[offset]
end for
offset := tid/k + (k − 1 − (tid mod k)) * N
(L⃗[offset], H⃗[offset], C⃗[offset]) := DeviceParallelMult(x⃗, y⃗, k, r)
                                   ▷ each thread computes one triple [l, h, c].
if tid < N then                    ▷ first N threads computing the final result.
    for (0 ≤ i < k) do   ▷ collecting the intermediate results from k − 1 adjacent threads
        offset := tid + i * N
        l⃗[i] := L⃗[offset]
        h⃗[i] := H⃗[offset]
        c⃗[i] := C⃗[offset]
    end for
    u⃗ := DeviceMultFinalResult(l⃗, h⃗, c⃗, k, r)          ▷ u⃗ := l⃗ + h⃗r + c⃗r^2
    for (0 ≤ i < k) do
        offset := tid + i * N
        U⃗[offset] := u⃗[i]
    end for
end if
return                                                  ▷ End of Kernel
```

**Algorithm 4.10** DeviceParallelMult($\vec{\mathtt{x}}, \vec{\mathtt{y}}, \mathtt{k}, \mathtt{r}$)

**input:**

- two positive integers $k$ and $r$ as specified in the introduction,
- vectors $\vec{\mathtt{x}}$ and $\vec{\mathtt{y}}$ representing two elements of $\mathbb{Z}/p\mathbb{Z}$ with $p = r^k + 1$.

**output:**

- vector $\vec{\mathtt{s}}$ representing an element of $\mathbb{Z}/p\mathbb{Z}$ with $p = r^k + 1$, storing result of intermediate additions ($\vec{\mathtt{s}} := \sum_{0 \le \mathtt{j} < \mathtt{k}} (\vec{\mathtt{x}}_{\mathtt{j}} * \vec{\mathtt{y}}_{\mathtt{k}-\mathtt{j}})$) in $\mathtt{k}$ digits of size of a machine-word.

**local:** vectors $\vec{\mathtt{s}} := [0, 0, 0], \vec{\mathtt{t}} := [0, 0, 0]$, each storing 3 digits of size of a machine-word.
**local:** `tid := blockIdx.x*blockSize.x+threadIdx.x`
**local:** $\mathtt{i} := (\mathtt{tid} \mod \mathtt{k})$ ▷ each thread computes the index of its corresponding digit.
$\vec{\mathtt{y}} :=$ DeviceMultPowR($\vec{\mathtt{y}}, \mathtt{i}, \mathtt{k}, \mathtt{r}$) ▷ $\vec{\mathtt{y}} := \vec{\mathtt{y}} r^i$
**for** $(0 \le \mathtt{j} < \mathtt{k})$ **do**
  $\vec{\mathtt{t}} :=$ DeviceIntermediateProduct($\vec{\mathtt{x}}[\mathtt{j}], \vec{\mathtt{y}}[\mathtt{k}-\mathtt{j}], \mathtt{k}, \mathtt{r}$)
  $\vec{\mathtt{s}} :=$ DeviceAddition($\vec{\mathtt{s}}, \vec{\mathtt{t}}, \mathtt{k}, \mathtt{r}$) ▷ computing addition only for 3 digits.
**end for**
**return** $\vec{\mathtt{s}}$

## 4.3 Profiling results

In this section, we present profiling results for our CUDA implementation of basic arithmetic operations in $\mathbb{Z}/p\mathbb{Z}$. Our code is optimized for prime $p = r^8 + 1$ with $r = 2^{63} + 2^{34}$. For each of four arithmetic operations, we compare performance metrics for the following variants:

1. functions for computing with non-transposed input vectors (based on the code developed by L. Chen[2]), and
2. our implementation of functions in this chapter for transposed input.

As we explained in Section 4.1, for computing each of arithmetic operations in $\mathbb{Z}/p\mathbb{Z}$, we assign one thread for computing each element of the final result.

**Analysis of functions for non-transposed data.** Assuming that the input is not transposed, a thread of the index `tid` reads the input vectors $\vec{\mathtt{X}}$ and $\vec{\mathtt{Y}}$ at the following

[2] http://faculty.ecnu.edu.cn/s/187/t/1487/main.jspy

memory addresses:

$$(\vec{\mathtt{X}}[8 * \mathtt{tid}], \ldots, \vec{\mathtt{X}}[8 * \mathtt{tid} + 7]),$$
$$(\vec{\mathtt{Y}}[8 * \mathtt{tid}], \ldots, \vec{\mathtt{Y}}[8 * \mathtt{tid} + 7]).$$

Therefore, based on what we described in Section 4.1.4, this implementation will be affected by strided access, because each warp is issuing more instructions for the same amount of data. As we explained in Chapter 2, GPU memory instructions take significantly more time than arithmetic instructions. Therefore, by issuing more memory requests, more warps will be waiting (stalling) for the data to arrive, and as a result, each warp has less data to process. Therefore, in case of addition, subtraction, and multiplication by powers of $r$, it is reasonable to expect very low values of IPC, because these algorithms do not re-use the input data at all. In comparison, for multiplication algorithm, we might see an increase in the value of IPC, because this algorithm is more arithmetic-intensive. At the same time, we expect to see a high value for the device memory utilization, provided that enough warps are scheduled on each streaming multiprocessor.

**Analysis of functions for transposed data.** For functions that compute the transposed input vector, we expect memory instruction overheads to be minimized. Moreover, we expect each thread to issue more arithmetic instructions, simply because the number of stalled cycles is much less than the other case. Therefore, for all of four arithmetic operations in $\mathbb{Z}/p\mathbb{Z}$, we expect the value of IPC to increase, and at the same time, number of instruction overheads to decrease. At the end, we would expect to see the following trends for addition, subtraction, and multiplication by power of r in $\mathbb{Z}/p\mathbb{Z}$:

1. the global memory throughput (for both load and store) will be closer to its peak, because as we explained in Chapter 2, that is a main attribute of memory bound kernels,
2. moreover, the value of IPC will be higher, because more warps are actively issuing arithmetic instructions at the same time, also, the value of IPC will be even higher (closer to its theoretical peak) for multiplication by powers of $r$.

**A note on the multiplication algorithm.** Similar to the other case, we expect to see a higher value of IPC for sequential multiplication. Therefore, for the non-transposed input functions, and for our sequential multiplication, we expect to see the value of IPC be of the same order. However, we expect the function for non-transposed data to have a lower memory store throughput (due to strided accesses). At the same time, for parallel multiplication, we expect to see a higher value of IPC, but a lower value for the memory store throughput. This expectation is reasonable, because parallel multiplication accesses to global memory (for storing the intermediate results) roughly three times more than other operations.

Profiling results for non-transposed addition, subtraction, and multiplication by powers of $r$ confirm our claims about the performance. For these operations, the global memory throughput (for both loading and storing) is around one third of its practical bandwidth.

Figures 4.7, 4.8, 4.9, 4.10, 4.11, 4.12, 4.13 present profiling results for four operations over $\mathbb{Z}/p\mathbb{Z}$, for randomly generated input vectors and with $N = 2^{17}$.

In each diagram, a specific metric for computing addition (represented by *Add*), subtraction (represented by *Sub*), multiplication by powers of radix (represented by *MultR*), and multiplication (represented by *Mult*) over $\mathbb{Z}/p\mathbb{Z}$ is presented by red and blue bars for non-transposed and transposed input, respectively. Profiling results are measured for the following metrics:

1. running time,
2. instruction overhead,
3. memory overhead,
4. the number of issued instructions per cycle (IPC),
5. the percentage of achieved occupancy,
6. the percentage of memory *load* efficiency, and
7. the percentage of memory *store* efficiency.

Finally, the profiling data has been collected on a NVIDIA Geforce-GTX760M card (hardware specifications are mentioned in Appendix B).

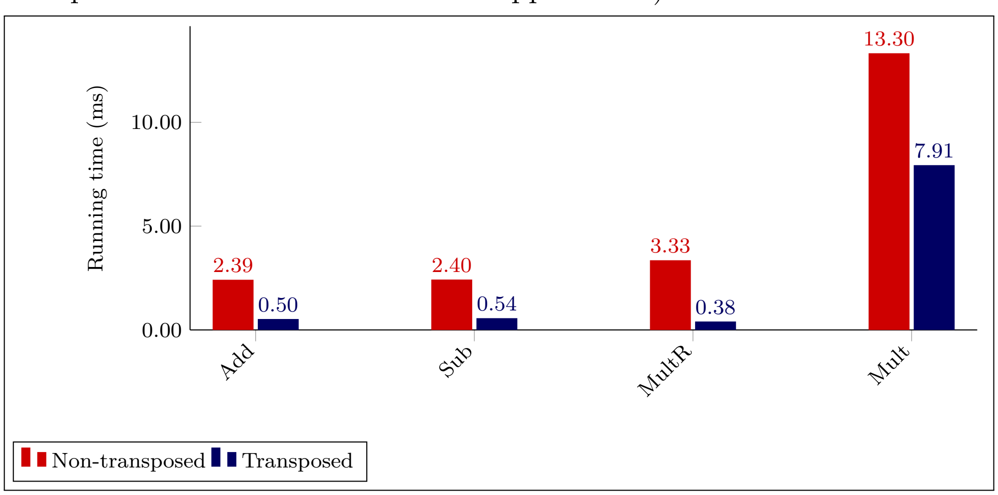


Figure 4.7: Diagram of running-time for $N = 2^{17}$.

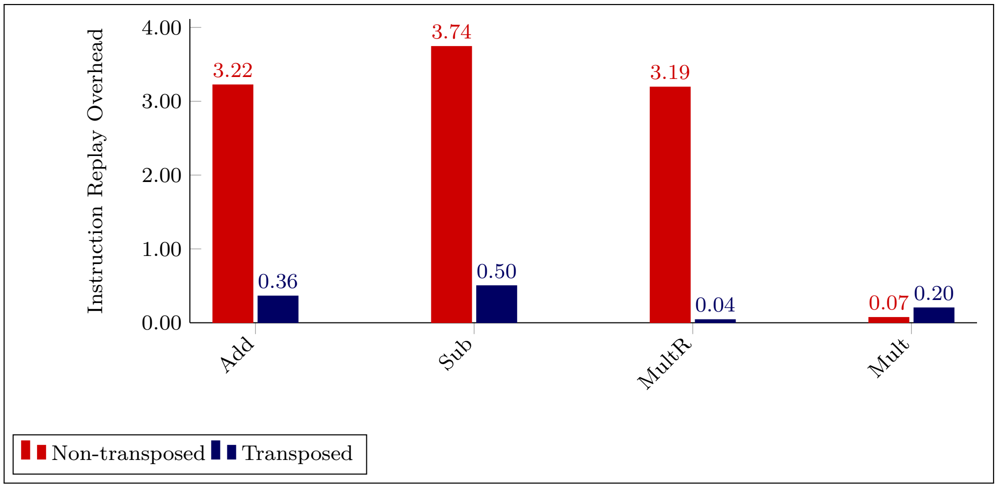


Figure 4.8: Diagram of instruction overhead for $N = 2^{17}$.

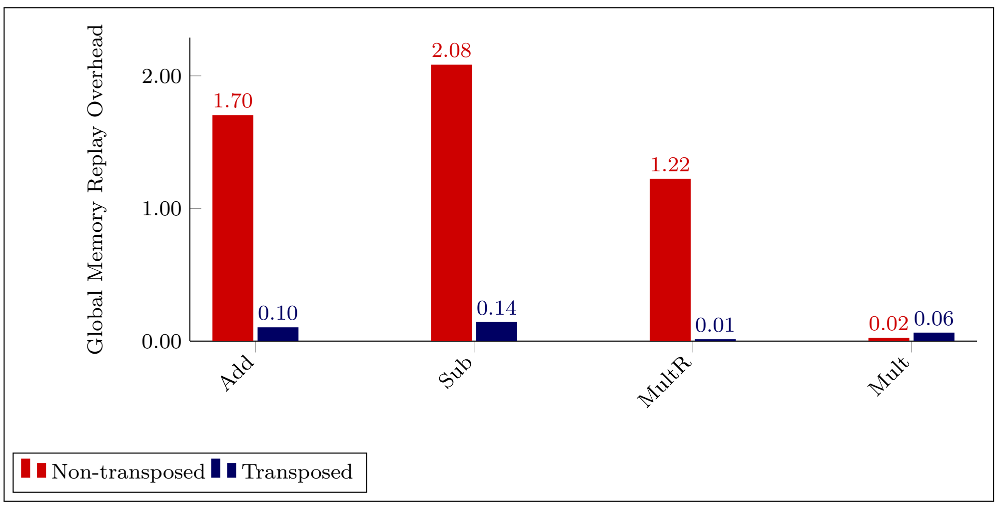


Figure 4.9: Diagram of memory overhead for $N = 2^{17}$.

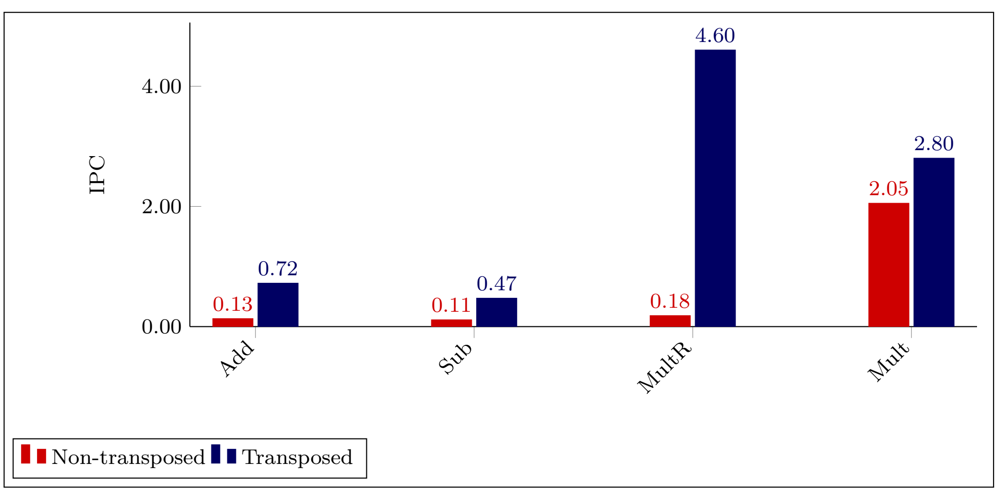


Figure 4.10: Diagram of IPC for $N = 2^{17}$.

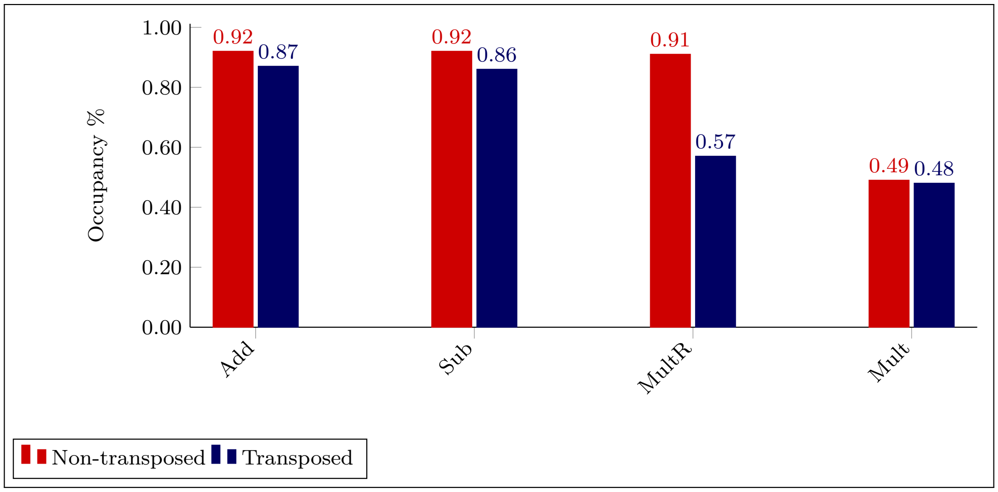


Figure 4.11: Diagram of occupancy percentage for $N = 2^{17}$.

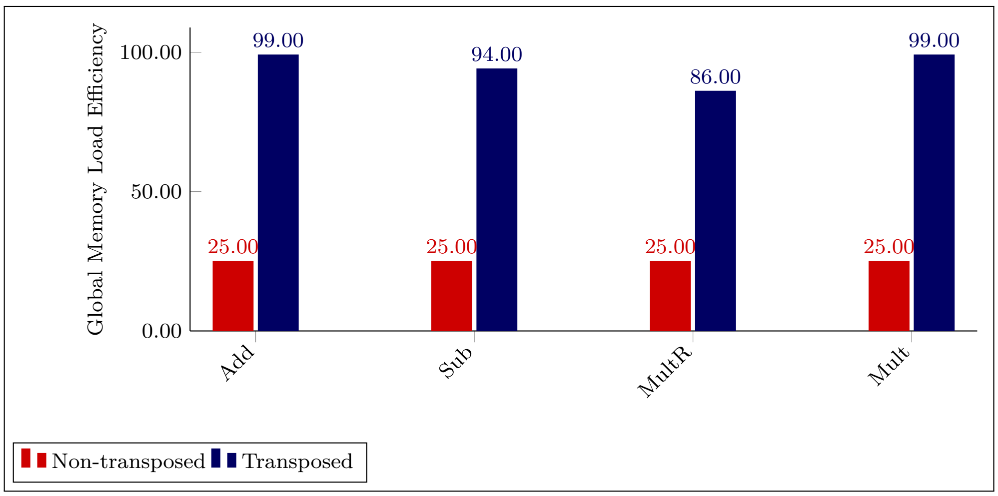


Figure 4.12: Diagram of memory load efficiency for $N = 2^{17}$.

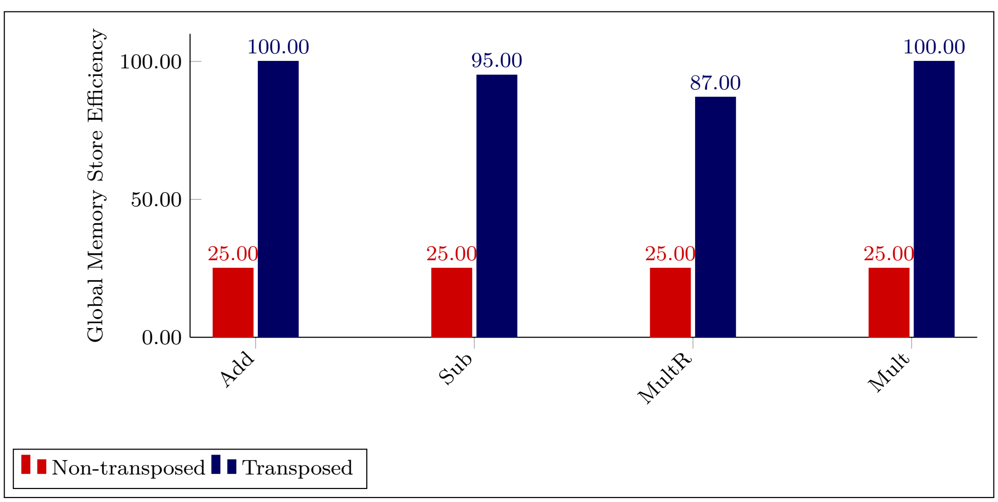


Figure 4.13: Diagram of memory store efficiency for $N = 2^{17}$.

# Chapter 5

# Stride Permutation on GPUs

Stride permutation is a basic part of the Cooley-Tukey FFT algorithm. Therefore, it is crucial to efficiently compute stride permutation on GPUs. In this chapter, first in Section 5.1, we explain how we can compute stride permutation on GPUs. Furthermore, in the same section, we discuss factors that affect the efficiency of computing stride permutations on GPUs. Finally, in Section 5.2, we have profiling results for our CUDA implementation of functions of this chapter.

## 5.1 Stride permutation

As we explained in Chapter 4, we would prefer to store the input data in a data structure that facilitates coalesced accesses to global memory. For this purpose, we assume that a vector of $N$ elements in $\mathbb{Z}/p\mathbb{Z}$ will be viewed as the transposition of a matrix $M$, with $N$ rows and $k$ columns, where $k$ is the power of radix in the prime $p = r^k + 1$.

For example, two elements $X_i$ and $X_j$ in $\mathbb{Z}/p\mathbb{Z}$ will be stored in the following way:

$$\vec{X}_i := (\vec{X}[i], \vec{X}[i + 1 * N], \ldots, \vec{X}[i + (k-1) * N])$$
$$\vec{X}_j := (\vec{X}[j], \vec{X}[j + 1 * N], \ldots, \vec{X}[j + (k-1) * N])$$

Therefore, every two adjacent digits of each element in $\mathbb{Z}/p\mathbb{Z}$ will be $N$ steps away from each other in memory. For example, the first and the second digits of $X_i$ are stored in $\vec{X}[i]$ and $\vec{X}[i + 1 * N]$, respectively. As we explained in Chapter 2, for a vector $\vec{x}$ with $mn$ elements in $\mathbb{Z}/p\mathbb{Z}$, stride permutation $L_m^{mn}$ computes the following permutation:

$$\vec{x}[in + j] \mapsto \vec{x}[i + mj],$$

with $0 \leq i < m$ and $0 \leq j < n$.

Based on this definition, if the input is an $n \times m$ matrix that is stored in the row-major layout, then this permutation is equivalent to the transposition:

$$L_m^{mn}(M_{n\times m}) = (M_{n\times m})^\top.$$

For example, for a vector $\vec{x} = (0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15)$ of 16 digits, we have the following stride permutations:

$$\begin{array}{rcl} L_2^{16}(\vec{x}) & = & (0, 2, 4, 6, 8, 10, 12, 14, 1, 3, 5, 7, 9, 11, 13, 15) \\ L_4^{16}(\vec{x}) & = & (0, 4, 8, 12, 1, 5, 9, 13, 2, 6, 10, 14, 3, 7, 11, 15) \\ L_8^{16}(\vec{x}) & = & (0, 8, 1, 9, 2, 10, 3, 11, 4, 12, 5, 13, 6, 14, 7, 15). \end{array}$$

Template StridePermutation presents a *naive* solution for computing the stride permutation $L_K^{KJ}$ for an input vector of $N$ elements in $\mathbb{Z}/p\mathbb{Z}$. Notice that this way of computing stride permutation has *low memory efficiency*, because it has accesses to memory addresses that are far away from each other (see Section 4.1.4). This pseudo-code is only given as an introductory example.

**Template** $\mathsf{StridePermutation}(\vec{\mathtt{X}}, \vec{\mathtt{Y}}, \mathtt{K}, \mathtt{N}, \mathtt{k})$

**input:**

- two positive integers $k$ and $r$ as specified in the introduction,
- a positive integer $\mathtt{K}$ representing the stride of the permutation,
- a positive integer $\mathtt{N}$,
- vector $\vec{\mathtt{X}}$ having $N$ elements of $\mathbb{Z}/p\mathbb{Z}$ with $p = r^k + 1$, thus storing $\mathtt{N} \times \mathtt{k}$ machine-words, viewed as the row-major layout of the transposition of a matrix $M_0$ with $N$ rows and $k$ columns.

**output:**

- vector $\vec{\mathtt{Y}}$ having $N$ elements of $\mathbb{Z}/p\mathbb{Z}$ with $p = r^k + 1$, thus storing $\mathtt{N} \times \mathtt{k}$ machine-words, viewed as the row-major layout of the transposition of a matrix $M_1$ with $N$ rows and $k$ columns, storing result of stride permutation such that $\vec{\mathtt{Y}} := L^{\mathtt{N}}_{\mathtt{K}}(\vec{\mathtt{X}})$.

```
local: offsetInput := 0, offsetOutput := 0
local: idxInput := 0, idxOutput := 0
local: J := N/K
for (0 ≤ j < J) do
    for (0 ≤ i < K) do
        idxInputElement := jK + i
        idxOutputElement := j + iJ
        for (0 ≤ c < k) do
            offsetInput := idxInputElement + c ∗ N
            offsetOutput := idxOutputElement + c ∗ N
            Y⃗[offsetOutput] := X⃗[offsetInput]
        end for
    end for
end for
 return Y
```

### 5.1.1 GPU kernels for stride permutation

As we will explain in Chapter 6, every step of computing a DFT requires permutations of different stride sizes. For example, for computing $\mathrm{DFT}_{16}$ based on $\mathrm{DFT}_2$, we should compute the following permutations:

$$L^4_2, L^8_2, L^{16}_2, L^8_4, L^{16}_8$$

Therefore, it is critical to have efficiently implemented functions for computing permutations of any stride sizes.

Computing stride permutations on GPUs relies on extensive use of shared memory and coalesced accesses to global memory. Basically, as we explained in Chapter 2, conflict-free accesses to shared emory have negligible cost. Therefore, using shared memory can reduce the cost of computing stride permutations. Stride permutations can be computed in the following way.

1. First, each thread block reads a portion of the input from global memory in a coalesced way.
2. In the next step, each thread block stores the data in shared memory.
3. Finally, each thread block writes the permutated data from shared memory to the output vector in global memory, in a coalesced way.

Assuming that our data is stored in the row-major layout, stride permutation is similar to the matrix transposition. As we explained in Chapter 2, matrix transposition is a memory bound kernel with very little arithmetic instructions to carry out. Therefore, the occupancy percentage will be a determining factor in the overall performance. Most significantly, the following parameters contribute to the overall percentage of achieved occupancy:

1. the size of a thread block on GPU, and
2. the the number of active warps per streaming multiprocessor.

We can compute stride permutations in one of the following ways:

1. by assigning multiple thread blocks for computing each stride permutation, or
2. by assigning exactly one thread block to each stride permutation.

For computing a permutation $L_K^{KJ}$, we have:

1. $b$ is the size of a one dimensional thread block,
2. $s$ is the total number of digits of size of a machine-word that can be stored in shared memory of a streaming multiprocessor,
3. $S$ is the number of streaming multiprocessors on the target GPU.

By this assumptions, the following assignments are possible.

$\mathbf{b} = \mathbf{K}$**.** In this case, we simply assign one thread block of $b$ threads for computing each permutation $L_K^{KJ}$. Algorithm 5.1 present a solution based on this assignment.

**b < K.** Again, we can assign one thread block of $b$ threads for computing each permutation $L_K^{KJ}$. Consequently, each thread block will transpose one sub-matrix of $s/b$ rows and $b$ columns at a time. In total, each thread block computes stride permutation for $T_0 := J/[s/b] \times K/b$ sub-matrices. On the other hand, we can assign $n = K/b$ thread blocks for computing each permutation $L_K^{KJ}$. In total, each thread computes stride permutation for $T_1 := J/[s/b] \times 1$ sub-matrices. As a result, there will be a higher utilization of streaming multiprocessors, and consequently. For a GPU with $S$ streaming multiprocessor, we define ratio $R$ in the following way:

$$R := \frac{\frac{T_0}{s}}{\frac{T_1}{s}} = \frac{K}{b}.$$

This implies that the second approach utilizes the streaming multiprocessor in a comparably more efficient way than the first approach. Algorithm 5.2 presents a solution for computing $L_K^{KJ}$ using $n = K/b$ thread blocks on GPUs.

**Algorithm 5.1** KernelBasePermutationSingleBlock($\vec{\mathtt{X}}, \vec{\mathtt{Y}}, \mathtt{K}, \mathtt{N}, \mathtt{k}, \mathtt{s}, \mathtt{r}$)

**input:**

- two positive integers $k$ and $r$ as specified in the introduction,
- a positive integer $\mathtt{K}$ representing the stride of the permutation,
- a positive integer $\mathtt{N}$,
- a positive integer $\mathtt{s}$ representing size of shared memory for each thread block,
- vector $\vec{\mathtt{X}}$ having $N$ elements of $\mathbb{Z}/p\mathbb{Z}$ with $p = r^k + 1$, thus storing $\mathtt{N} \times \mathtt{k}$ machine-words, viewed as the row-major layout of the transposition of a matrix $M_0$ with $N$ rows and $k$ columns.

**output:**

- vector $\vec{\mathtt{Y}}$ having $N$ elements of $\mathbb{Z}/p\mathbb{Z}$ with $p = r^k + 1$, thus storing $\mathtt{N} \times \mathtt{k}$ machine-words, viewed as the row-major layout of the transposition of a matrix $M_1$ with $N$ rows and $k$ columns, storing result of stride permutation such that $\vec{\mathtt{Y}} := L^{\mathtt{N}}_{\mathtt{K}}(\vec{\mathtt{X}})$.

```
local: tid := blockIdx.x*blockSize.x+threadIdx.x
local: offsetDigit:=0
local: j:=threadIdx.x
local: J:=N/K
local: h := s/k
local: c:=0
local: offsetBlock := blockIdx.x * K * J
__shared__ shmem[s]    ▷ allocating shared memory, which is visible to all threads of a the same thread block.
for (0 ≤ c < k) do
    offsetDigit := c * N
    for (0 ≤ r < J/h) do
        for (0 ≤ i < h) do
            shmem[j + i * K] = X⃗[offsetDigit + offsetBlock + j + i * k]
            __syncThreads
            Y⃗[offsetDigit + offsetBlock + j * h + i] = shmem[j * h + i]
        end for
    end for
end for
return                                                  ▷ End of Kernel
```

**Algorithm 5.2** KernelBasePermutationMultipleBlocks($\vec{X}, \vec{Y}, K, N, k, s, r$)

**input:**

- two positive integers $k$ and $r$ as specified in the introduction,
- a positive integer K representing the stride of the permutation,
- a positive integer N,
- a positive integer s representing size of shared memory for each thread block,
- vector $\vec{X}$ having $N$ elements of $\mathbb{Z}/p\mathbb{Z}$ with $p = r^k + 1$, thus storing $N \times k$ machine-words, viewed as the row-major layout of the transposition of a matrix $M_0$ with $N$ rows and $k$ columns.

**output:**

- vector $\vec{Y}$ having $N$ elements of $\mathbb{Z}/p\mathbb{Z}$ with $p = r^k + 1$, thus storing $N \times k$ machine-words, viewed as the row-major layout of the transposition of a matrix $M_1$ with $N$ rows and $k$ columns, storing result of stride permutation such that $\vec{Y} := L^{N}_{K}(\vec{X})$.

```
local: tid := blockIdx.x*blockSize.x+threadIdx.x
local: offsetDigit:=0
local: j:=threadIdx.x
local: b:=blockSize.x                          ▷ b :=Dimension of a 1D thread block
local: J:=N/K
local: h := s/k
local: c:=0
local: offsetPermutation:=0
local: offsetBlock:=0
offsetBlock := blockIdx.x * J * b
__shared__ shmem[s]
for (0 ≤ c < k) do
    offsetDigit := c * N
    for (0 ≤ r < J/h) do
        for (0 ≤ i < h) do
            shmem[j + i * K] = X[offsetDigit + offsetBlock + j + i * k]
            __syncThreads
            Y[offsetDigit + offsetBlock + j * h + i] = shmem[j * h + i]
        end for
    end for
end for
return                                         ▷ End of Kernel
```

**Size of a thread block** Basically, stride permutation is a memory bound kernel on GPUs, as it computes a number of arithmetic instructions, and the overall performance of the implementation is determined by the way that we have access to the memory. Therefore, as we explained in Chapter 2, it is crucial to have a high occupancy percentage to hide the data latency. Consequently, we should choose the size of a thread block, $b$ by considering three objectives:

1. maximizing the throughput of reading from global memory,
2. maximizing the throughput of writing to global memory, and finally
3. maximizing the occupancy percentage to hide the data latency.

Assume that each streaming multiprocessor can store $s$ digits of size of a machine-wordon its shared memory. Therefore, each thread block can compute stride permutation for a sub-matrix of $s/b$ rows and $b$ columns. Based on what we explained in the the previous section, the larger values of $b$ will restrict us to read less columns from the input. At a given moment, each block will read $s/b$ rows of size $b$, which is equivalent of $s/b$ columns of size $b$ in output. Our goal is to maximize $s/b$ and at the same time, choose $b$ large enough to achieve a high value of occupancy on each of streaming multiprocessors. Our experimental results demonstrate that for $p = r^8 + 1$ and for $s = \frac{2^{15}}{2^3}$ digits, we achieve the best results for threads blocks of 128 threads and 256 threads, respectively (see Section 5.2).

### 5.1.2 Host entry point for permutation kernels

Finally, we need to have a `host` function as an entry point for initializing the data and invoking the GPU kernel functions. Algorithm 5.3 presents a host function that will initialize data, then will choose a suitable GPU kernel for computing stride permutation. Moreover, we assume that grids and thread blocks are one dimensional.

**Algorithm 5.3** HostGeneralStridePermutation ($\vec{\mathtt{X}}, \vec{\mathtt{Y}}, \mathtt{K}, \mathtt{N}, \mathtt{k}, \mathtt{s}, \mathtt{r}, \mathtt{b}$)

**input:**

- two positive integers $k$ and $r$ as specified in the introduction,
- a positive integer $\mathtt{K}$ representing the stride of the permutation,
- a positive integer $\mathtt{N}$,
- a positive integer $\mathtt{s}$ representing size of shared memory for each thread block,
- a positive $b$ integer representing size of a 1D thread block,
- vector $\vec{\mathtt{X}}$ having $N$ elements of $\mathbb{Z}/p\mathbb{Z}$ with $p = r^k + 1$, thus storing $\mathtt{N} \times \mathtt{k}$ machine-words, viewed as the row-major layout of the transposition of a matrix $M_0$ with $N$ rows and $k$ columns.

**output:**

- vector $\vec{\mathtt{Y}}$ having $N$ elements of $\mathbb{Z}/p\mathbb{Z}$ with $p = r^k + 1$, thus storing $\mathtt{N} \times \mathtt{k}$ machine-words, viewed as the row-major layout of the transposition of a matrix $M_1$ with $N$ rows and $k$ columns, storing result of stride permutation such that $\vec{\mathtt{Y}} := L^{\mathtt{N}}_{\mathtt{K}}(\vec{\mathtt{X}})$.

▷ In either case, initializing 1D grid of dimension $\mathtt{N}/\mathtt{b}$

**if** $b < K$ **then**
  KernelBasePermutationMultipleBlocks$<<<\ \mathtt{N}/\mathtt{b}, \mathtt{b}\ >>>(\vec{\mathtt{X}}, \vec{\mathtt{Y}}, \mathtt{K}, \mathtt{N}, \mathtt{k}, \mathtt{s}, \mathtt{r})$
**else if** $b = k$ **then**
  KernelBasePermutationSingleBlock$<<<\ \mathtt{N}/\mathtt{b}, \mathtt{b}\ >>>(\vec{\mathtt{X}}, \vec{\mathtt{Y}}, \mathtt{K}, \mathtt{N}, \mathtt{k}, \mathtt{s}, \mathtt{r})$
**end if**
**return** ▷ End of Kernel

## 5.2 Profiling results

In this section, we have the profiling results for the CUDA implementation of Algorithms 5.1 and 5.2, respectively.

Figure 5.1 shows the result of profiling for computing $L_K^{KJ}$ with $K = 256$ and $J = 4096$. For thread blocks of size $b = 256$, and shared memory of size $s = 2^{12}$ digits of size of a machine-word, this implementation assigns 8 thread blocks for computing each stride permutation.

Also, Figure 5.2 shows the profiling result for the implementation that assigns one thread block for computing the permutation $L_K^{KJ}$, with $K = 16$ and $J = 2^{16}$.

The profiling results are measured for the following metrics:

1. the percentage of achieved occupancy,

2. the total number of issued instructions per cycle (IPC),
3. instruction overhead,
4. throughput of loading data from global memory,
5. throughput of storing data to global memory, and
6. the efficiency percentage for accessing global memory.

As the final note, we have collected the profiling data on a NVIDIA Geforce-GTX760M card (hardware specifications are mentioned in Appendix B).

```
Invocations                               Metric Name                        Metric Description         Min         Max         Avg
Device "GeForce GTX 760M (0)"
 Kernel: kernel_permutation_256_general_permutated_v0(__int64, __int64*, __int64*, __int64*)
          1                        achieved_occupancy                        Achieved Occupancy    0.124812    0.124812    0.124812
          1                                       ipc                              Executed IPC    0.085657    0.085657    0.085657
          1                      inst_replay_overhead               Instruction Replay Overhead    0.710638    0.710638    0.710638
          1                            gst_throughput                   Global Store Throughput  3.9570GB/s  3.9570GB/s  3.9570GB/s
          1                            gld_throughput                    Global Load Throughput  3.9609GB/s  3.9609GB/s  3.9609GB/s
          1                            gld_efficiency             Global Memory Load Efficiency      99.93%      99.93%      99.93%
          1                            gst_efficiency            Global Memory Store Efficiency     100.00%     100.00%     100.00%
```

Figure 5.1: Profiling results for stride permutation $L_K^{KJ}$ for $K = 256$ and $J = 4096$.

```
Invocations                               Metric Name                        Metric Description         Min         Max         Avg
Device "GeForce GTX 760M (0)"
 Kernel: kernel_permutation_16_permutated(__int64*, __int64*)
          1                        achieved_occupancy                        Achieved Occupancy    0.248948    0.248948    0.248948
          1                                       ipc                              Executed IPC    0.087653    0.087653    0.087653
          1                      inst_replay_overhead               Instruction Replay Overhead    1.915645    1.915645    1.915645
          1                            gst_throughput                   Global Store Throughput  3.9794GB/s  3.9794GB/s  3.9794GB/s
          1                            gld_throughput                    Global Load Throughput  3.9872GB/s  3.9872GB/s  3.9872GB/s
          1                            gld_efficiency             Global Memory Load Efficiency      99.85%      99.85%      99.85%
          1                            gst_efficiency            Global Memory Store Efficiency     100.00%     100.00%     100.00%
```

Figure 5.2: Profiling results for stride permutation $L_K^{KJ}$ for $K = 16$ and $J = 2^{16}$.

# Chapter 6

# Big Prime Field FFT on GPUs

In this chapter, we explain how we can compute FFT for vectors of elements in $\mathbb{Z}/p\mathbb{Z}$ on GPUs. First, in Section 6.1, we have a quick review of the Cooley-Tukey FFT algorithm. Then, in Section 6.2, we explain an algorithm for computing multiplication by *twiddle factors* on GPUs. Furthermore, in Section 6.3, we explain how by using *six-step recursive FFT*, we can compute FFT through a base-case formula that is faster in practice. Next, in Section 6.4, we explain how we can compute the FFT for vectors of any length in $\mathbb{Z}/p\mathbb{Z}$. Finally, in Section 6.5, we have profiling results for CUDA implementation of algorithms of this chapter.

## 6.1 Cooley-Tukey FFT

As we explained in Chapter 2, for computing the FFT for a vector of $N = KJ$ elements in $\mathbb{Z}/p\mathbb{Z}$, and for $\omega^N = 1$, the Cooley-Tukey FFT algorithm factorizes the computation in the following way:

$$\mathrm{DFT}_N = (\mathrm{DFT}_K \otimes I_J) D_{K,J} (I_K \otimes \mathrm{DFT}_J) L_K^N.$$

In this notation, $D_{K,J}$ represents the multiplication by the powers of $\omega$. Moreover, the *diagonal twiddle matrix* $D_{K,J}$ is defined as

$$D_{K,J} = \bigoplus_{j=0}^{K-1} \mathrm{diag}(1, \omega_i^j, \ldots, \omega_i^{j(J-1)}).$$

In practice, The Cooley-Tukey FFT algorithm is not a suitable choice for implementation on GPUs, mostly because of the way that it accesses the memory. Therefore, we need an equivalent equation which is more suitable for structure of GPUs. That is, we must have an equation that can efficiently exploit *block parallelism* of GPUs. In terms of tensor notation, block parallelism can be realized by tensor products of the form $I_J \otimes \mathrm{DFT}_K$, and therefore, we should find a solution to convert our computations to the mentioned form. For this purpose, we use the *six-step recursive FFT algorithm* [10], which is expressed in the following way:

$$\mathrm{DFT}_N = L_K^N (I_J \otimes \mathrm{DFT}_K) L_J^N D_{K,J} (I_K \otimes \mathrm{DFT}_J) L_K^N.$$

By this formula, we can further expand the left part $I_J \otimes \mathrm{DFT}_K$ to reduce all computations to a base-case $\mathrm{DFT}_K$. Accordingly, by having an efficient implementation for computing $\mathrm{DFT}_K$, we can have a high performance implementation of the FFT.

## 6.2 Multiplication by twiddle factors

Multiplications by twiddle factors can be computed using the multiplication algorithm of Section 4.2.5. However, as we explained in Chapter 3, one of our goals is to use the cheap multiplications by powers of radix, as much as we can. Therefore, for computing $\mathrm{DFT}_N$ based on $\mathrm{DFT}_K$, we compute twiddle factor multiplications in a different way. Basically, for computing $\mathrm{DFT}_{K^e}$ by $\mathrm{DFT}_K$, we require the multiplications of the form $D_{K,K^{e-s}}$ where $\omega_i = \omega^{K^{(s-1)}}$ $(1 \leq s < e)$. Also, as we know, $\omega^N = r^{2k}$. Therefore, by choosing $K = 2k$, result of $y := x * \omega^{i(N/K)+j}$ can be computed in the following way:

1. first, $y := x * \omega^{i(N/K)} = x * r^i$ which can be computed by multiplication algorithm of Section 4.2.4, then,
2. $y := y * \omega^j$ which can be computed by the multiplication algorithm of Section 4.2.5.

Therefore, for computing the multiplication by the twiddle factors, we should only compute multiplications for $\omega^j$ with $0 < j < N/K$. In this case, we can pre-compute and store the powers of $\omega$ up to $\omega^{N/K-1}$. Conclusively, for computing $\mathrm{DFT}_{K^e}$, we need to store powers of $\omega$ up to $\omega^{K^{e-1}-1}$.

In practice, we store the pre-computed powers of $\omega$ either in the global memory, or in the texture memory (preferred) of GPUs. Similar to other arithmetic operations, we assign exactly one thread for computing element of final result of multiplication by powers of $\omega$. Algorithm 6.1 presents the solution for computing $D_{K,K^{e-s}}$ using $K^{(e-s)}$ threads on GPUs.

**Algorithm 6.1** KernelTwiddleMultiplication($\vec{\mathtt{X}}, \vec{\Omega}, \mathtt{N}, \mathtt{K}, \mathtt{k}, \mathtt{s}, \mathtt{r}$)

**input:**

- two positive integers $k$ and $r$ as specified in the introduction,
- a positive integer $\mathtt{N}$,
- a positive integer $\mathtt{s}$ representing the step of twiddle factor multiplication,
- vector $\vec{\mathtt{X}}$ having $N$ elements of $\mathbb{Z}/p\mathbb{Z}$ with $p = r^k + 1$, thus storing $\mathtt{N} \times \mathtt{k}$ machine-words, viewed as the row-major layout of the transposition of a matrix $M_0$ with $N$ rows and $k$ columns,
- vector $\vec{\Omega}$ having $K^{(e-1)}$ elements of $\mathbb{Z}/p\mathbb{Z}$ with $p = r^k + 1$, thus storing $K^{(e-1)} \times k$ machine-words, viewed as the row-major layout of the transposition of a matrix $M_1$ with $K^{(e-1)}$ rows and $k$ columns.

**output:**

- vector $\vec{\mathtt{X}}$ storing result of twiddle factor multiplication $(D_{K,K^{(e-s)}})$

**local:** $\mathtt{i} := \mathtt{tid}/(\mathtt{K}^{\mathtt{e}} - \mathtt{s})$

**local:** $\mathtt{j} := \mathtt{tid} \mod (\mathtt{K}^{\mathtt{e}} - \mathtt{s})$

**local:** $\mathtt{v} := (\mathtt{i} * \mathtt{j})/(\mathtt{K}^{\mathtt{e}} - 1)$

**local:** $\mathtt{c} := (\mathtt{i} * \mathtt{j}) \mod (\mathtt{K}^{\mathtt{e}} - 1)$

**local:** vectors $\vec{\mathtt{x}}$, $\vec{\mathtt{y}}$, $\vec{\mathtt{u}}$, $\vec{\mathtt{l}}$, $\vec{\mathtt{h}}$, $\vec{\mathtt{c}}$ each storing $\mathtt{k}$ digits of size of a machine-word, all digits initially set to 0.

**local:** $\mathtt{offset} := 0$

**for** $(0 \leq \mathtt{i} < \mathtt{k})$ **do**
  $\mathtt{offset} := \mathtt{tid} + \mathtt{i} * \mathtt{N}$
  $\vec{\mathtt{x}}[\mathtt{i}] := \vec{\mathtt{X}}[\mathtt{offset}]$ ▷ Loading digit of the index $\mathtt{i}$ from element $\vec{\mathtt{X}}_{\mathtt{tid}}$.
**end for**
**for** $(0 \leq \mathtt{i} < \mathtt{k})$ **do**
  $\mathtt{offset} := \mathtt{c} + \mathtt{i} * \mathtt{N}$
  $\vec{\mathtt{y}}[\mathtt{i}] := \vec{\Omega}[\mathtt{offset}]$ ▷ Loading digit of the index $\mathtt{i}$ from element $\vec{\Omega}_{\mathtt{c}}$.
**end for**
$\vec{\mathtt{x}} :=$ DeviceCyclicShift($\vec{\mathtt{x}}, \mathtt{v}, \mathtt{k}, \mathtt{r}$)
$[\vec{\mathtt{l}}, \vec{\mathtt{h}}, \vec{\mathtt{c}}] :=$ DeviceSequentialMult($\vec{\mathtt{x}}, \vec{\mathtt{y}}, \mathtt{k}, \mathtt{r}$) ▷ Each thread computing $k$ digits.
$\vec{\mathtt{u}} :=$ DeviecMultFinalResult($\vec{\mathtt{l}}, \vec{\mathtt{h}}, \vec{\mathtt{c}}, \mathtt{k}, \mathtt{r}$)
**for** $(0 \leq \mathtt{i} < \mathtt{k})$ **do**
  $\mathtt{offset} := \mathtt{tid} + \mathtt{i} * \mathtt{N}$
  $\vec{\mathtt{X}}[\mathtt{offset}] := \vec{\mathtt{x}}[\mathtt{i}]$ ▷ Storing digit of the index $\mathtt{i}$ to element $\vec{\mathtt{X}}_{\mathtt{tid}}$.
**end for**
  **return** $\vec{\mathtt{X}}$

## 6.3 Implementation of the base-case DFT-K

In this section, we explain algorithms for computing $\mathrm{DFT}_N$ based on $\mathrm{DFT}_K$. First, we explain how by using the six-step FFT algorithm, we can further decompose the base-case $\mathrm{DFT}_K$. Moreover, we will describe an algorithm for computing $\mathrm{DFT}_2$ for two elements of $\mathbb{Z}/p\mathbb{Z}$ on GPUs. Finally, we present algorithms for computing $\mathrm{DFT}_16$ based on 8 intermediate steps that only compute $\mathrm{DFT}_2$, or permutations, or multiplications by powers of radix.

### 6.3.1 Expanding DFT-K based on six-step FFT

By using the six-step FFT equation, we can further expand the $\mathrm{DFT}_K$ until we reach to a point where all of computations are reduced to $\mathrm{DFT}_2$. For example, for $K = 2^\ell$, we can derive the following equations:

$$\begin{array}{lllllllll}
\mathrm{DFT}_{2^\ell} & = & L_2^{2^\ell} & (I_{2^{(\ell-1)}} \otimes DFT_2) & L_{2^{(\ell-1)}}^{2^\ell} & D_{K,2^{(\ell-1)}} & (I_2 \otimes DFT_{2^{(\ell-1)}}) & L_2^{2^\ell}, \\
\mathrm{DFT}_{2^{(\ell-1)}} & = & L_2^{2^{(\ell-1)}} & (I_{2^{(\ell-2)}} \otimes DFT_2) & L_{2^{(\ell-2)}}^{2^{(\ell-1)}} & D_{K,2^{(\ell-2)}} & (I_2 \otimes DFT_{2^{(\ell-2)}}) & L_2^{2^{(\ell-1)}}, \\
\vdots & & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\
\mathrm{DFT}_4 & = & L_2^4 & (I_2 \otimes DFT_2) & L_2^4 & D_{K,2} & (I_2 \otimes DFT_2) & L_2^4.
\end{array}$$

Also, by using the algorithms of Chapter 5, we can compute stride permutations of any sizes on GPUs. However, for vectors that store $K$ elements of $\mathbb{Z}/p\mathbb{Z}$, if the value of $K$ is not large, we compute permutation in a different way. Basically, performing a permutation on a vector, changes the order of elements in that vector, or in other words, permutation changes the index of each element. Therefore, we can pre-compute index of each element after permutation is performed. In other sense, instead of moving the data inside the memory, we only compute the memory address that will be modified by a permutation.

### 6.3.2 Implementation of DFT-2

For two arbitrary elements $X_0$ and $X_1$ in $\mathbb{Z}/p\mathbb{Z}$, $\mathrm{DFT}_2$ is computed in the following way:

$$\mathrm{DFT}_2(X_0, X_1) := (X_0 + X_1, X_0 - X_1).$$

Algorithm 6.2 presents the solution for computing $\mathrm{DFT}_2$ for two elements of $\mathbb{Z}/p\mathbb{Z}$ on GPUs. For this algorithm, we assign exactly one thread for computing result of $\mathrm{DFT}_2$ for two elements of $\mathbb{Z}/p\mathbb{Z}$.

**Algorithm 6.2** DeviceDFT2($\vec{\mathtt{X}}, \mathtt{i}, \mathtt{j}, \mathtt{N}, \mathtt{k}, \mathtt{r}$)

**input:**

- two positive integers $k$ and $r$ as specified in the introduction,
- a positive integer $\mathtt{N}$,
- positive integers $\mathtt{i}$ and $\mathtt{j}$, representing the indexes of two elements of input vector, namely $\vec{\mathtt{X}}_\mathtt{i}$ and $\vec{\mathtt{X}}_\mathtt{j}$ with $\mathtt{i} \neq \mathtt{j}$,
- vector $\vec{\mathtt{X}}$ having $N$ elements of $\mathbb{Z}/p\mathbb{Z}$ with $p = r^k + 1$, thus storing $\mathtt{N} \times \mathtt{k}$ machine-words, viewed as the row-major layout of the transposition of a matrix $M_0$ with $N$ rows and $k$ columns.

**output:**

- vector $\vec{\mathtt{X}}$ with result of $\mathrm{DFT}_2$ computed for two digits $\vec{\mathtt{X}}_\mathtt{i}$ and $\vec{\mathtt{X}}_\mathtt{j}$ and stored in $\vec{\mathtt{X}}$ such that $(\vec{\mathtt{X}}_\mathtt{i}, \vec{\mathtt{X}}_\mathtt{j}) := \mathrm{DFT}_2(\vec{\mathtt{X}}_\mathtt{i}, \vec{\mathtt{X}}_\mathtt{j})$

**local:** `c:=0, offset:=0`

**local:** `tid := blockIdx.x*blockSize.x+threadIdx.x`

**local:** vectors $\vec{\mathtt{A}}, \vec{\mathtt{B}}, \vec{\mathtt{S}}_0, \vec{\mathtt{S}}_1$ each storing $k$ digits of size of a machine-word

**for** $(0 \leq \mathtt{c} < \mathtt{k})$ **do** $\triangleright\ \vec{\mathtt{A}} := \vec{\mathtt{X}}_\mathtt{i}$

  $\mathtt{offset} := \mathtt{i} + \mathtt{c} * \mathtt{N}$

  $\vec{\mathtt{A}}[\mathtt{c}] := \vec{\mathtt{X}}[\mathtt{offset}]$

**end for**

**for** $(0 \leq \mathtt{c} < \mathtt{k})$ **do** $\triangleright\ \vec{\mathtt{B}} := \vec{\mathtt{X}}_\mathtt{j}$

  $\mathtt{offset} := \mathtt{j} + \mathtt{c} * \mathtt{N}$

  $\vec{\mathtt{B}}[\mathtt{c}] := \vec{\mathtt{X}}[\mathtt{offset}]$

**end for**

$\vec{\mathtt{S}}_0[0 : \mathtt{k} - 1] := \mathsf{DeviceAddition}(\vec{\mathtt{A}}, \vec{\mathtt{B}}, \mathtt{k}, \mathtt{k}, \mathtt{r})$ $\triangleright\ \vec{\mathtt{S}}_0 := \vec{\mathtt{A}} + \vec{\mathtt{B}}$

$\vec{\mathtt{S}}_1[0 : \mathtt{k} - 1] := \mathsf{DeviceSubtraction}(\vec{\mathtt{A}}, \vec{\mathtt{B}}, \mathtt{k}, \mathtt{k}, \mathtt{r})$ $\triangleright\ \vec{\mathtt{S}}_1 := \vec{\mathtt{A}} - \vec{\mathtt{B}}$

**for** $(0 \leq \mathtt{c} < \mathtt{k})$ **do** $\triangleright\ \vec{\mathtt{X}}_\mathtt{i} := \vec{\mathtt{S}}_0$

  $\mathtt{offset} := \mathtt{i} + \mathtt{c} * \mathtt{N}$

  $\vec{\mathtt{X}}[\mathtt{offset}] := \vec{\mathtt{S}}_0[\mathtt{c}]$

**end for**

**for** $(0 \leq \mathtt{c} < \mathtt{k})$ **do** $\triangleright\ \vec{\mathtt{X}}_\mathtt{j} := \vec{\mathtt{S}}_1$

  $\mathtt{offset} := \mathtt{j} + \mathtt{c} * \mathtt{N}$

  $\vec{\mathtt{X}}[\mathtt{offset}] := \vec{\mathtt{S}}_1[\mathtt{c}]$

**end for**

**return**

### 6.3.3 Computing DFT-16 based on DFT-2

For the prime $p = r^8 + 1$, we choose $K = 2k = 16$ as the size of our base-case DFT.

In the rest of this section, we describe how we can expand the base-case $\mathrm{DFT}_{16}$ to a number of base-case $\mathrm{DFT}_2$ computations. We must take into account that for computing the multiplication by twiddle factors, each base-case DFT needs different powers of $\omega$ in the following way:

1. $\mathrm{DFT}_{16}$ needs $\omega_0 = \omega^{N/K} = r$,
2. $\mathrm{DFT}_8$ needs $\omega_1 = \omega^{(N/K)^2} = r^2$,
3. and finally, $\mathrm{DFT}_4$ needs $\omega_2 = \omega^{(N/K)^4} = r^4$.

Expanding $DFT_{16}$ based on the six-step FFT algorithm results in the following sequence of equations:

$$\begin{array}{rcl} DFT_{16} & = & L_2^{16}(I_8 \otimes DFT_2)L_8^{16}D_{2,8}^{16}(I_2 \otimes DFT_8)L_2^{16}, \\ DFT_8 & = & L_2^{8}(I_4 \otimes DFT_2)L_4^{8}D_{2,4}^{8}(I_2 \otimes DFT_4)L_2^{8}, \\ DFT_4 & = & L_2^{4}(I_2 \otimes DFT_2)L_2^{4}D_{2,2}^{4}(I_2 \otimes DFT_2)L_2^{4}, \end{array}$$

which can be re-written as:

$$\begin{gathered} DFT_{16} = L_2^{16}(I_8 \otimes DFT_2)L_8^{16}D_{2,8}^{16}, \\ (I_2 \otimes L_2^{8}(I_4 \otimes DFT_2)L_4^{8}D_{2,4}^{8}(I_2 \otimes L_2^{4}(I_2 \otimes DFT_2)L_2^{4}D_{2,2}^{4}(I_2 \otimes DFT_2)L_2^{4}L_2^{8}L_2^{16}. \end{gathered}$$

Furthermore, the following twiddle factor multiplications are needed:

$$\begin{array}{rcl} D_{2,8}^{16} & = & (1,1,1,1,1,1,1,1,\omega_0,\omega_0^1,\omega_0^2,\omega_0^3,\omega_0^4,\omega_0^5,\omega_0^6,\omega_0^7), \\ D_{2,4}^{8} & = & (1,1,1,1,\omega_1^0,\omega_1^1,\omega_1^2,\omega_1^3), \\ D_{2,2}^{4} & = & (1,1,\omega_2,\omega_2^2), \end{array}$$

which are equivalent of

$$\begin{array}{rcl} D_{2,8}^{16} & = & (1,1,1,1,1,1,1,1,r^0,r^1,r^2,r^3,r^4,r^5,r^6,r^7), \\ D_{2,4}^{8} & = & (1,1,1,1,r^0,r^2,r^4,r^6), \\ D_{2,2}^{4} & = & (1,1,r^0,r^4). \end{array}$$

We compute the base-case $\mathrm{DFT}_{16}$ on a vector of 16 elements of $\mathbb{Z}/p\mathbb{Z}$, namely, $\vec{M}$:

$$\vec{M} = (X_0, X_1, X_2, X_3, X_4, X_5, X_6, X_7, X_8, X_9, X_{10}, X_{11}, X_{12}, X_{13}, X_{14}, X_{15})$$

The computation of $\mathrm{DFT}_{16}$ on $\vec{M}$ can be broken into eight steps.

**Step 1**

In this step, the following sequence of of permutations are needed:

$$
\begin{array}{lcl}
\vec{M} & = & (X_0, X_1, X_2, X_3, X_4, X_5, X_6, X_7, X_8, X_9, X_{10}, X_{11}, X_{12}, X_{13}, X_{14}, X_{15}), \\
L_2^{16}\vec{M} & = & (X_0, X_2, X_4, X_6, X_8, X_{10}, X_{12}, X_{14}, X_1, X_3, X_5, X_7, X_9, X_{11}, X_{13}, X_{15}), \\
(I_2 \otimes L_2^8)L_2^{16}\vec{M} & = & (X_0, X_4, X_8, X_{12}, X_2, X_6, X_{10}, X_{14}, X_1, X_5, X_9, X_{13}, X_3, X_7, X_{11}, X_{15}), \\
M_0 & = & (I_2 \otimes I_2 \otimes L_2^4)(I_2 \otimes L_2^8)L_2^{16}\vec{M} \\
M_0 & = & (X_0, X_8, X_4, X_{12}, X_2, X_{10}, X_6, X_{14}, X_1, X_9, X_5, X_{13}, X_3, X_{11}, X_7, X_{15}).
\end{array}
$$

After that, $DFT_2$ should be computed for every two elements in $\vec{M}_0$. The final result of this stop will be stored in $\vec{M}_1$.

$$
\begin{array}{lcl}
\vec{M}_1 & := & I_2 \otimes I_2 \otimes I_2 \otimes DFT_2(M_0) \\
 & := & I_8 \otimes DFT_2(M_0) \\
 & := & [\mathrm{DFT}_2(X_0, X_8), \mathrm{DFT}_2(X_4, X_{12}), \mathrm{DFT}_2(X_2, X_{10}), \mathrm{DFT}_2(X_6, X_{14}), \\
 & & \mathrm{DFT}_2(X_1, X_9), \mathrm{DFT}_2(X_5, X_{13}), \mathrm{DFT}_2(X_3, X_{11}), \mathrm{DFT}_2(X_7, X_{15})].
\end{array}
$$

Algorithm 6.3 presents the pseudo-code for computing this step.

**Algorithm 6.3** DeviceDFT16Step1($\vec{\mathtt{X}}, \mathtt{N}, \mathtt{k}, \mathtt{r}$)

**input:**

- two positive integers $k$ and $r$ as specified in the introduction,
- a positive integer N,
- vector $\vec{\mathtt{X}}$ having $N$ elements of $\mathbb{Z}/p\mathbb{Z}$ with $p = r^k + 1$, thus storing $\mathtt{N} \times \mathtt{k}$ machine-words, viewed as the row-major layout of the transposition of a matrix $M_0$ with $N$ rows and $k$ columns.

**output:**

- Step 1 of DFT-16 for $\vec{\mathtt{X}}$.

**local:** $\mathtt{t} := 0, \mathtt{offset} := 0, \mathtt{idx}_0 := 0, \mathtt{idx}_1 := 0$

**local:** `tid := blockIdx.x*blockSize.x+threadIdx.x`

$\mathtt{t} := \mathtt{tid} \mod (8)$

$\mathtt{offset} := (\mathtt{tid}/16) * 8$ ▷ each thread computes two elements, 8 threads compute 16 elements.

$\mathtt{idx}_0 := \mathtt{t} + \mathtt{offset}$

$\mathtt{idx}_1 := \mathtt{t} + \mathtt{offset} + 8$

DeviceDFT2($\vec{\mathtt{X}}, \mathtt{idx}_0, \mathtt{idx}_1, \mathtt{N}, \mathtt{k}, \mathtt{r}$)

**Step 2**

In this step, the following twiddle factor multiplications for $\omega_2 = r^4$ should be computed:

$$\begin{aligned}\vec{M}_2 &:= (I_2 \otimes I_2)D_{2,2}(M_1) := (I_4 \otimes D_{2,2})(M_1)\\ &:= [(X_0, X_8, X_4, X_{12} * r^4), (X_2, X_{10}, X_6, X_{14} * r^4),\\ &\quad (X_1, X_9, X_5, X_{13} * r^4), (X_3, X_{11}, X_7, X_{15} * r^4)].\end{aligned}$$

The final result of this step will be stored in vector $\vec{M}_2$. Algorithm 6.4 presents the pseudo-code for computing this step.

**Algorithm 6.4** DeviceDFT16Step2($\vec{\mathtt{X}}, \mathtt{N}, \mathtt{k}, \mathtt{r}$)

**input:**

- two positive integers $k$ and $r$ as specified in the introduction,
- a positive integer $\mathtt{N}$,
- vector $\vec{\mathtt{X}}$ having $N$ elements of $\mathbb{Z}/p\mathbb{Z}$ with $p = r^k + 1$, thus storing $\mathtt{N} \times \mathtt{k}$ machine-words, viewed as the row-major layout of the transposition of a matrix $M_0$ with $N$ rows and $k$ columns.

**output:**

- Step 2 of DFT-16 for $\vec{\mathtt{X}}$.

**local:** $\mathtt{t} := 0, \mathtt{offset} := 0, \mathtt{idx}_0 := 0, \mathtt{idx}_1 := 0$

**local:** `tid := blockIdx.x*blockSize.x+threadIdx.x`

**local:** vector $\vec{\mathtt{A}}$ representing one element of $\mathbb{Z}/p\mathbb{Z}$ with $p = r^k + 1$, storing temporary values in $\mathtt{k}$ digits of size of a machine-word.

**local:** $\mathtt{s} := 0$

**local:** $\mathtt{t} := \mathtt{tid} \mod (8)$

```
if t < 4 then                                      ▷ power of radix in r^s
    s := 0                                         ▷ s := 0 for first four threads
else
    s := 4                                         ▷ s := 4 for last four threads
end if
offset := (tid/16) * 8
idx_0 := t + offset                                ▷ every thread computes two elements
for (0 ≤ c < k) do
    A[c] := X[idx_0 + c * permutationStride]
end for
A[0 : k − 1] :=DeviceCyclicShift(X, s, k, r)
for (0 ≤ c < k) do
    X[idx_0 + c * permutationStride] := A[c]
end for
```

### Step 3

In this step, the following stride permutation should be computed for $\vec{M}_2$. The result of this permutation will be stored in vector $\vec{M}_3$.

$$\begin{aligned} \vec{M}_3 &:= I_2 \otimes L_2^4 \vec{M}_2 \\ &:= (X_0, X_4, X_8, X_{12}, X_2, X_6, X_{10}, X_{14}, X_1, X_5, X_9, X_{13}, X_3, X_7, X_{11}, X_{15}). \end{aligned}$$

Then, $DFT_2$ should be computed for every two elements of $\vec{M}_3$.

$$\begin{aligned}\vec{M}_4 &:= DFT_2\vec{M}_3\\ &:= [DFT_2(X_0,X_4), DFT_2(X_8,X_{12}), DFT_2(X_2,X_6), DFT_2(X_{10},X_{14}),\\ &\quad DFT_2(X_1,X_5), DFT_2(X_9,X_{13}), DFT_2(X_3,X_7), DFT_2(X_{11},X_{15})].\end{aligned}$$

The final result of this step will be stored in $\vec{M}_3$. Algorithm 6.5 presents the pseudo-code for computing this step.

**Algorithm 6.5** DeviceDFT16Step3($\vec{\mathtt{X}}$, N, k, r)

**input:**

- two positive integers $k$ and $r$ as specified in the introduction,
- a positive integer N,
- vector $\vec{\mathtt{X}}$ having $N$ elements of $\mathbb{Z}/p\mathbb{Z}$ with $p = r^k + 1$, thus storing $\mathtt{N} \times \mathtt{k}$ machine-words, viewed as the row-major layout of the transposition of a matrix $M_0$ with $N$ rows and $k$ columns.

**output:**

- Step 3 of DFT-16 for $\vec{\mathtt{X}}$.

**local:** $\mathtt{t} := 0, \mathtt{offset} := 0, \mathtt{idx}_0 := 0, \mathtt{idx}_1 := 0$

**local:** tid := blockIdx.x*blockSize.x+threadIdx.x

$\mathtt{t} := \mathtt{tid} \mod (8)$

$\mathtt{offset} := (\mathtt{tid}/16) * 8$

**if** $\mathtt{t} > 4$ **then**

  $\mathtt{t} := \mathtt{t} + 4$

**end if** ▷ every thread computes two elements

$\mathtt{idx}_0 := \mathtt{t} + \mathtt{offset}$

$\mathtt{idx}_1 := \mathtt{t} + \mathtt{offset} + 8$

DeviceDFT2($\vec{\mathtt{X}}, \mathtt{idx}_0, \mathtt{idx}_1, \mathtt{N}, \mathtt{k}, \mathtt{r}$)

### Step 4

In this step, first, the following permutation should be computed for $\vec{M}_4$. The result of this permutation will be stored in vector $\vec{M}_5$.

$$\begin{aligned}\vec{M}_5 &:= I_2 \otimes L_2^4\vec{M}_4\\ &:= [X_0,X_8,X_4,X_{12},X_2,X_{10},X_6,X_{14},X_1,X_9,X_5,X_{13},X_3,X_{11},X_7,X_{15}].\end{aligned}$$

Then, the following twiddle factor multiplication for $\omega_1 = r^2$ will be computed for $\vec{M}_5$.

$$\begin{aligned}\vec{M}_6 : &= D_{2,4}(\vec{M}_5)\\ &:= [(X_0, X_8, X_4, X_{12}), (X_2, X_{10}, X_6, X_{14}),\\ &\quad (X_1, X_9 * r^0, X_5, X_{13} * r^2), (X_3, X_{11} * r^4, X_7, X_{15} * r^6)].\end{aligned}$$

The final result of this step will be stored in vector $\vec{M}_6$. Algorithm 6.6 presents the pseudo-code for computing this step.

**Algorithm 6.6** DeviceDFT16Step4($\vec{\texttt{X}}, \texttt{N}, \texttt{k}, \texttt{r}$)

**input:**

- two positive integers $k$ and $r$ as specified in the introduction,
- a positive integer N,
- vector $\vec{\texttt{X}}$ having $N$ elements of $\mathbb{Z}/p\mathbb{Z}$ with $p = r^k + 1$, thus storing $\texttt{N} \times \texttt{k}$ machine-words, viewed as the row-major layout of the transposition of a matrix $M_0$ with $N$ rows and $k$ columns.

**output:**

- Step 4 of DFT-16 for $\vec{\texttt{X}}$.

**local:** $\texttt{t} := 0, \texttt{offset} := 0, \texttt{idx}_0 := 0, \texttt{idx}_1 := 0$

**local:** tid := blockIdx.x*blockSize.x+threadIdx.x

**local:** vector $\vec{\texttt{A}}$ representing one element of $\mathbb{Z}/p\mathbb{Z}$ with $p = r^k + 1$, storing temporary values in k digits of size of a machine-word.

**local:** $\texttt{t} := \texttt{tid} \mod (8)$

**local:** List:=[0,0,0,0,0,4,2,6]

**local:** $\texttt{s} := \texttt{List}[\texttt{t}]$

**if** $\texttt{t} > 4$ **then**

  $\texttt{t} := 2 * \texttt{t} + 1$

**end if**

$\texttt{offset} := (\texttt{tid}/16) * 8$

$\texttt{idx}_0 := \texttt{t} + \texttt{offset}$ ▷ every thread computes two elements.

**for** $(0 \leq \texttt{c} < \texttt{k})$ **do**

  $\vec{\texttt{A}}[\texttt{c}] := \vec{\texttt{X}}[\texttt{idx}_0 + \texttt{c} * \texttt{permutationStride}]$

**end for**

$\texttt{A}[0 : \texttt{k} - 1]$ :=DeviceCyclicShift(X, s, k, r)

**for** $(0 \leq \texttt{c} < \texttt{k})$ **do**

  $\vec{\texttt{X}}[\texttt{idx}_0 + \texttt{c} * \texttt{permutationStride}] := \vec{\texttt{A}}[\texttt{c}]$

**end for**

### Step 5

In this step, the following permutation will be computed on $\vec{M}_6$. The result of this permutation will be stored in vector $\vec{M}_7$.

$$\begin{aligned}\vec{M}_7 &:= I_2 \otimes L_4^8 \vec{M}_6\\ &:= (X_0, X_2, X_8, X_{10}, X_4, X_6, X_{12}, X_{14}, X_1, X_3, X_9, X_{11}, X_5, X_7, X_{13}, X_{15}).\end{aligned}$$

Then, $\mathrm{DFT}_2$ will be computed for every two elements of $\vec{M}_7$.

$$\begin{aligned}\vec{M}_8 &:= DFT_2 \vec{M}_7\\ &:= (DFT_2(X_0, X_2), DFT_2(X_8, X_{10}), DFT_2(X_4, X_6), DFT_2(X_{12}, X_{14}),\\ &\quad\; DFT_2(X_1, X_3), DFT_2(X_9, X_{11}), DFT_2(X_5, X_7), DFT_2(X_{13}, X_{15})).\end{aligned}$$

The final result of this step will be stored $\vec{M}_8$. Algorithm 6.7 presents the pseudo-code for computing this step.

**Algorithm 6.7** DeviceDFT16Step5($\vec{\mathtt{X}}, \mathtt{N}, \mathtt{k}, \mathtt{r}$)

**input:**

- two positive integers $k$ and $r$ as specified in the introduction,
- a positive integer N,
- vector $\vec{\mathtt{X}}$ having $N$ elements of $\mathbb{Z}/p\mathbb{Z}$ with $p = r^k + 1$, thus storing $\mathtt{N} \times \mathtt{k}$ machine-words, viewed as the row-major layout of the transposition of a matrix $M_0$ with $N$ rows and $k$ columns.

**output:**

- Step 5 of DFT-16 for $\vec{\mathtt{X}}$.

```
local: t := 0, offset := 0, idx_0 := 0, idx_1 := 0
local: tid := blockIdx.x*blockSize.x+threadIdx.x
t := tid mod (8)
t := 2 * t − (t mod (2))
offset := (tid/16) * 8 ▷ Every thread computes two elements, 8 threads compute 16 elements
idx_0 := t + offset
idx_1 := t + offset + 2
DeviceDFT2(X, idx_0, idx_1, N, k, r)
```

**Step 6**

In this step, first, the following permutation will be computed for $\vec{M}_8$:

$$\begin{aligned}\vec{M}_9 &:= I_2 \otimes L_2^8 \vec{M}_8 \\ &:= (X_0, X_8, X_4, X_{12}, X_2, X_{10}, X_6, X_{14}, X_1, X_9, X_5, X_{13}, X_3, X_{11}, X_7, X_{15}).\end{aligned}$$

The result of this permutation will be stored in vector $\vec{M}_9$. Then, the following twiddle factor multiplication for $\omega_0 = r$ will be computed on $\vec{M}_9$:

$$\begin{aligned}\vec{M}_{10} &:= D_{2,8}(\vec{M}_9) \\ &:= [(X_0, X_8, X_4, X_{12}, X_2, X_{10}, X_6, X_{14}), \\ &\quad (X_1 * r^0, X_9 * r^1, X_5 * r^2, X_{13} * r^3, X_3 * r^4, X_{11} * r^5, X_7 * r^6, X_{15} * r^7)].\end{aligned}$$

The final result of this step will be stored in vector $\vec{M}_{10}$. Algorithm 6.8 presents the pseudo-code for computing this step.

**Algorithm 6.8** DeviceDFT16Step6($\vec{X}$, N, k, r)

**input:**

- two positive integers $k$ and $r$ as specified in the introduction,
- a positive integer N,
- vector $\vec{X}$ having $N$ elements of $\mathbb{Z}/p\mathbb{Z}$ with $p = r^k + 1$, thus storing $\mathtt{N} \times \mathtt{k}$ machine-words, viewed as the row-major layout of the transposition of a matrix with $N$ rows and $k$ columns.

**output:**

- Step 6 of DFT-16 for $\vec{X}$.

**local:** $\mathtt{t} := 0, \mathtt{offset} := 0, \mathtt{idx}_0 := 0, \mathtt{idx}_1 := 0$

**local:** tid := blockIdx.x∗blockSize.x+threadIdx.x

**local:** vector $\vec{A}$ representing one element of $\mathbb{Z}/p\mathbb{Z}$ with $p = r^k + 1$, storing temporary values in k digits of size of a machine-word.

**local:** $\mathtt{t} := \mathtt{tid} \mod (8)$

**local:** List:=[0,4,2,6,1,5,3,7]

$\mathtt{t} := 2 * \mathtt{t} + 1$

**local:** $\mathtt{s} := \mathtt{List}[\mathtt{t}]$

$\mathtt{offset} := (\mathtt{tid}/16) * 8$

$\mathtt{idx}_0 := \mathtt{t} + \mathtt{offset}$ ▷ Every thread computes two elements

**for** $(0 \leq \mathtt{c} < \mathtt{k})$ **do**

  $\vec{A}[\mathtt{c}] := \vec{X}[\mathtt{idx}_0 + \mathtt{c} * \mathtt{permutationStride}]$

**end for**

$\vec{A}[0 : \mathtt{k} - 1]$ :=DeviceCyclicShift($\vec{X}$, s, k, r)

**for** $(0 \leq \mathtt{c} < \mathtt{k})$ **do**

  $\vec{X}[\mathtt{idx}_0 + \mathtt{c} * \mathtt{permutationStride}] := \vec{A}[\mathtt{c}]$

**end for**

### Step 7

In this step, first, the following permutation will be computed for $\vec{M}_{10}$:

$$\begin{aligned}\vec{M}_{11} &:= L_8^{16}\vec{M}_{10} \\ &:= (X_0, X_1, X_8, X_9, X_4, X_5, X_{12}, X_{13}, X_2, X_3, X_10, X_{11}, X_6, X_7, X_{14}, X_{15}).\end{aligned}$$

The result of this step will be stored in vector $\vec{M}_{11}$. Then, $\mathrm{DFT}_2$ will be computed for every two elements of $\vec{M}_1 1$ in the following way:

$$\begin{aligned}\vec{M}_{12} &:= I_8 \otimes DFT_2 \vec{M}_{11}\\ &:= (X_0, X_1, X_8, X_9, X_4, X_5, X_{12}, X_{13}, X_2, X_3, X_{10}, X_{11}, X_6, X_7, X_{14}, X_{15}).\end{aligned}$$

The final result of this step will be stored in $\vec{M}_{12}$. Algorithm 6.9 presents the pseudo-code for computing this step.

**Algorithm 6.9** DeviceDFT16Step7($\vec{\mathtt{X}}$, N, k, r)

**input:**

- two positive integers $k$ and $r$ as specified in the introduction,
- a positive integer N,
- vector $\vec{\mathtt{X}}$ having $N$ elements of $\mathbb{Z}/p\mathbb{Z}$ with $p = r^k + 1$, thus storing $\mathtt{N} \times \mathtt{k}$ machine-words, viewed as the row-major layout of the transposition of a matrix $M_0$ with $N$ rows and $k$ columns.

**output:**

- Step 7 of DFT-16 for $\vec{\mathtt{X}}$.

**local:** $\mathtt{t} := 0, \mathtt{offset} := 0, \mathtt{idx}_0 := 0, \mathtt{idx}_1 := 0$

**local:** tid := blockIdx.x*blockSize.x+threadIdx.x

$\mathtt{t} := \mathtt{tid} \mod (8)$

$\mathtt{t} := 2 * \mathtt{t}$

$\mathtt{offset} := (\mathtt{tid}/16) * 8$ ▷ Every thread computes two elements, 8 threads compute 16 elements

$\mathtt{idx}_0 := \mathtt{t} + \mathtt{offset}$

$\mathtt{idx}_1 := \mathtt{t} + \mathtt{offset} + 1$

DeviceDFT2($\vec{\mathtt{X}}$, $\mathtt{idx}_0$, $\mathtt{idx}_1$, N, k, r)

### Step 8

This is the final step for computing $\mathrm{DFT}_{16}$ on $\vec{M}$. In this step, only the following permutation will be computed for $\vec{M}_{13}$:

$$\begin{aligned}\vec{M}_{13} &:= L_2^{16} \vec{M}_{10}\\ &:= (X_0, X_8, X_4, X_{12}, X_2, X_{10}, X_6, X_{14}, X_1, X_9, X_5, X_{13}, X_3, X_{11}, X_7, X_{15}).\end{aligned}$$

The final result will be stored in $\vec{M}_{13}$. Algorithm 6.10 presents the pseudo-code for computing this step.

**Algorithm 6.10** DeviceDFT16Step8($\vec{\mathtt{X}}, \mathtt{N}, \mathtt{k}, \mathtt{r}$)

**input:**

- two positive integers $k$ and $r$ as specified in the introduction,
- a positive integer $\mathtt{N}$,
- vector $\vec{\mathtt{X}}$ having $N$ elements of $\mathbb{Z}/p\mathbb{Z}$ with $p = r^k + 1$, thus storing $\mathtt{N} \times \mathtt{k}$ machine-words, viewed as the row-major layout of the transposition of a matrix $M_0$ with $N$ rows and $k$ columns.

**output:**

- Step 8 of DFT-16 for $\vec{\mathtt{X}}$.

**local:** $\mathtt{t} := 0, \mathtt{offset} := 0, \mathtt{idx}_0 := 0, \mathtt{idx}_1 := 0$

**local:** `tid := blockIdx.x*blockSize.x+threadIdx.x`

**local:** vector $\vec{\mathtt{A}}$ representing one element of $\mathbb{Z}/p\mathbb{Z}$ with $p = r^k + 1$, storing temporary values in $\mathtt{k}$ digits of size of a machine-word.

**local:** $\mathtt{t} := \mathtt{tid} \mod (8)$

**local:** $\mathtt{List} := [0, 2, -2, 0, -7, -5, -9, -7, 7, 9, 5, 7, 0, 2, -2, 0]$

**local:** $\mathtt{s} := \mathtt{List}[\mathtt{t}]$

$\mathtt{offset} := (\mathtt{tid}/16) * 8$

$\mathtt{idx}_0 := \mathtt{t} + \mathtt{offset}$

$\mathtt{idx}_1 := \mathtt{idx}_0 + \mathtt{s}$

**if** $s > 0$ **then**

  **for** $(0 \leq \mathtt{c} < \mathtt{k})$ **do**

    $\mathtt{tmp} := \vec{\mathtt{X}}[\mathtt{idx}_0 + \mathtt{c} * \mathtt{permutationStride}]$

    $\vec{\mathtt{X}}[\mathtt{idx}_0 + \mathtt{c} * \mathtt{permutationStride}] := \vec{\mathtt{X}}[\mathtt{idx}_1 + \mathtt{c} * \mathtt{permutationStride}]$

    $\vec{\mathtt{X}}[\mathtt{idx}_1 + \mathtt{c} * \mathtt{permutationStride}] := \mathtt{tmp}$

  **end for**

**end if**

$\mathtt{idx}_0 := \mathtt{idx}_0 + 1$

$\mathtt{idx}_1 := \mathtt{idx}_0 + \mathtt{s}$

**if** $s > 0$ **then**

  **for** $(0 \leq \mathtt{c} < \mathtt{k})$ **do**

    $\mathtt{tmp} := \vec{\mathtt{X}}[\mathtt{idx}_0 + \mathtt{c} * \mathtt{permutationStride}]$

    $\vec{\mathtt{X}}[\mathtt{idx}_0 + \mathtt{c} * \mathtt{permutationStride}] := \vec{\mathtt{X}}[\mathtt{idx}_1 + \mathtt{c} * \mathtt{permutationStride}]$

    $\vec{\mathtt{X}}[\mathtt{idx}_1 + \mathtt{c} * \mathtt{permutationStride}] := \mathtt{tmp}$

  **end for**

**end if**

Algorithm 6.11 presents the pseudo-code of the kernel for computing $\mathrm{DFT}_{16}$ for a vector of 16 elements in $\mathbb{Z}/p\mathbb{Z}$. We assign exactly 8 threads for computing this kernel, because every thread will compute one $\mathrm{DFT}_2$ or one multiplication by power of radix.

**Algorithm 6.11** KernelBaseDFT16AllSteps($\vec{\mathtt{X}}, \mathtt{N}, \mathtt{k}, \mathtt{r}$)

**input:**

- two positive integers $k$ and $r$ as specified in the introduction,
- a positive integer $\mathtt{N}$,
- vector $\vec{\mathtt{X}}$ having $N$ elements of $\mathbb{Z}/p\mathbb{Z}$ with $p = r^k + 1$, thus storing $\mathtt{N} \times \mathtt{k}$ machine-words, viewed as the row-major layout of the transposition of a matrix $M_0$ with $N$ rows and $k$ columns.

**output:**

- vector $\vec{\mathtt{X}}$, storing final result of DFT-16 on every sub-vector of 16 elements in $\vec{\mathtt{X}}$ ($\vec{\mathtt{X}} := I_{N/16} \otimes \mathrm{DFT}_{16}(\vec{\mathtt{X}})$).

**local:** $\mathtt{t} := 0, \mathtt{offset} := 0, \mathtt{idx}_0 := 0, \mathtt{idx}_1 := 0$

**local:** `tid := blockIdx.x*blockSize.x+threadIdx.x`

DeviceDFT16Step1($\vec{\mathtt{X}}, \mathtt{N}, \mathtt{k}, \mathtt{r}$)

DeviceDFT16Step2($\vec{\mathtt{X}}, \mathtt{N}, \mathtt{k}, \mathtt{r}$)

DeviceDFT16Step3($\vec{\mathtt{X}}, \mathtt{N}, \mathtt{k}, \mathtt{r}$)

DeviceDFT16Step4($\vec{\mathtt{X}}, \mathtt{N}, \mathtt{k}, \mathtt{r}$)

DeviceDFT16Step5($\vec{\mathtt{X}}, \mathtt{N}, \mathtt{k}, \mathtt{r}$)

DeviceDFT16Step6($\vec{\mathtt{X}}, \mathtt{N}, \mathtt{k}, \mathtt{r}$)

DeviceDFT16Step7($\vec{\mathtt{X}}, \mathtt{N}, \mathtt{k}, \mathtt{r}$)

DeviceDFT16Step8($\vec{\mathtt{X}}, \mathtt{N}, \mathtt{k}, \mathtt{r}$)

## 6.4 Host entry point for computing DFT

In this section, first, we explain how we compute the FFT-$K^2$ using the base-case $DFT_K$ of previous section. Also, we present a general algorithm for computing the FFT for $N = K^e$ based on $DFT_K$.

### 6.4.1 FFT-$K^2$

For $N = K^2$, the six-step recursive FFT algorithm can be expressed in the following way:

$$DFT_{K^2} = L_K^{K^2}(I_K \otimes DFT_K)L_K^{K^2} D_{K,K}(I_K \otimes DFT_K)L_K^{K^2}$$

Algorithm 6.12 presents the solution for computing $\mathrm{DFT}_{K^2}$ using the base-case $\mathrm{DFT}_K$.

**Algorithm 6.12** HostDFTK2($\vec{\mathtt{X}}, \vec{\Omega}, \mathtt{N}, \mathtt{K}, \mathtt{k}, \mathtt{s}, \mathtt{r}, \mathtt{b}$)

**input:**

- two positive integers $k$ and $r$ as specified in the introduction,
- a positive integer $\mathtt{N}$,
- a positive integer $\mathtt{s}$ representing the step of twiddle factor multiplication,
- a positive integer $b$ representing the size of a one dimensional thread block,
- vector $\vec{\mathtt{X}}$ having $N$ elements of $\mathbb{Z}/p\mathbb{Z}$ with $p = r^k + 1$, thus storing $\mathtt{N} \times \mathtt{k}$ machine-words, viewed as the row-major layout of the transposition of a matrix $M_0$ with $N$ rows and $k$ columns,
- vector $\vec{\Omega}$ having $K^{N/K}$ elements of $\mathbb{Z}/p\mathbb{Z}$ with $p = r^k + 1$, thus storing $K^{N/K} \times k$ machine-words, viewed as the row-major layout of the transposition of a matrix $M_1$ with $K^{N/K}$ rows and $k$ columns.

**output:**

- vector $\vec{\mathtt{X}}$ storing result of FFT-$K^2$ ($\vec{\mathtt{X}} := \mathrm{DFT}_{K^2}(\vec{\mathtt{X}})$)

**local:** vector $\vec{\mathtt{B}}$ of size $N$

HostGeneralStridePermutation($\vec{\mathtt{X}}, \vec{\mathtt{Y}}, \mathtt{K}, \mathtt{N}, \mathtt{k}, \mathtt{s}, \mathtt{b}$)

$\vec{\mathtt{X}}[0 : \mathtt{kN} - 1] := \vec{\mathtt{Y}}[0 : \mathtt{kN} - 1]$

KernelBaseDFTKAllSteps($\tilde{\mathtt{X}}, \mathtt{N}, \mathtt{K}, \mathtt{r}$)

KernelTwiddleMultiplication($\vec{\mathtt{X}}, \vec{\Omega}, \vec{\mathtt{L}}, \vec{\mathtt{H}}, \vec{\mathtt{C}}, \mathtt{N}, \mathtt{k}, \mathtt{r}$)

HostGeneralStridePermutation($\vec{\mathtt{X}}, \vec{\mathtt{Y}}, \mathtt{K}, \mathtt{N}, \mathtt{k}, \mathtt{s}, \mathtt{b}$)

$\vec{\mathtt{X}}[0 : \mathtt{kN} - 1] := \vec{\mathtt{Y}}[0 : \mathtt{kN} - 1]$

KernelBaseDFTKAllSteps($\vec{\mathtt{X}}, \mathtt{N}, \mathtt{k}, \mathtt{r}$)

HostGeneralStridePermutation($\vec{\mathtt{X}}, \vec{\mathtt{Y}}, \mathtt{K}, \mathtt{N}, \mathtt{k}, \mathtt{s}, \mathtt{b}$)

$\vec{\mathtt{X}}[0 : \mathtt{kN} - 1] := \vec{\mathtt{Y}}[0 : \mathtt{kN} - 1]$

**return** $\vec{\mathtt{X}}$

### 6.4.2 FFT-general based on K

Algorithm 6.13 presents an algorithm for computing FFTs for vectors of $N = K^e$ elements in $\mathbb{Z}/p\mathbb{Z}$.

**Algorithm 6.13** HostDFTGeneral($\vec{\mathtt{X}}, \vec{\Omega}, \mathtt{N}, \mathtt{K}, \mathtt{k}, \mathtt{s}, \mathtt{r}, \mathtt{b}$)

**input:**

- two positive integers $k$ and $r$ as specified in the introduction,
- a positive integer $\mathtt{N}$,
- a positive integer $\mathtt{s}$ representing the step of twiddle factor multiplication,
- a positive integer $b$ representing the size of a one dimensional thread block,
- vector $\vec{\mathtt{X}}$ having $N$ elements of $\mathbb{Z}/p\mathbb{Z}$ with $p = r^k + 1$, thus storing $\mathtt{N} \times \mathtt{k}$ machine-words, viewed as the row-major layout of the transposition of a matrix $M_0$ with $N$ rows and $k$ columns,
- vector $\vec{\Omega}$ having $K^{N/K}$ elements of $\mathbb{Z}/p\mathbb{Z}$ with $p = r^k + 1$, thus storing $K^{N/K} \times k$ machine-words, viewed as the row-major layout of the transposition of a matrix $M_1$ with $K^{N/K}$ rows and $k$ columns.

**output:**

- vector $\vec{\mathtt{X}}$ storing result of FFT-N ($\vec{\mathtt{X}} := \mathrm{DFT}_N(\vec{\mathtt{X}})$)

**local:** $\mathtt{m} := \mathtt{e}$ where $N = K^e$

**local:** $\mathtt{j} = 0$

**if** $\mathtt{e} \mod 2 = 1$ **then**

  HostGeneralStridePermutation($\vec{\mathtt{X}}, \vec{\mathtt{Y}}, \mathtt{K}^1, \mathtt{N}, \mathtt{k}, \mathtt{s}, \mathtt{b}$)

  $\vec{\mathtt{X}}[0 : \mathtt{kN} - 1] := \vec{\mathtt{Y}}[0 : \mathtt{kN} - 1]$

  $\mathtt{m} := \mathtt{m} - 1$

**end if**

**for** ($0 \leq \mathtt{i} < \mathtt{m}$ by 2) **do**

  HostDFTK2($\vec{\mathtt{X}}, \vec{\Omega}, \mathtt{N}, \mathtt{K}, \mathtt{k}, \mathtt{s}, \mathtt{r}$)

  KernelTwiddleMultiplication($\vec{\mathtt{X}}, \vec{\Omega}, \mathtt{N}, \mathtt{K}, \mathtt{k}, \mathtt{s} := 2, \mathtt{r}$)

  HostGeneralStridePermutation($\vec{\mathtt{X}}, \vec{\mathtt{Y}}, \mathtt{K}^2, \mathtt{N}, \mathtt{k}, \mathtt{s}, \mathtt{b}$)

  $\vec{\mathtt{X}}[0 : \mathtt{kN} - 1] := \vec{\mathtt{Y}}[0 : \mathtt{kN} - 1]$

  HostDFTK2($\vec{\mathtt{X}}, \vec{\Omega}, \mathtt{N}, \mathtt{K}, \mathtt{k}, \mathtt{s}, \mathtt{r}$)

  HostGeneralStridePermutation($\vec{\mathtt{X}}, \vec{\mathtt{Y}}, \mathtt{K}^2, \mathtt{N}, \mathtt{k}, \mathtt{s}, \mathtt{b}$)

  $\vec{\mathtt{X}}[0 : \mathtt{kN} - 1] := \vec{\mathtt{Y}}[0 : \mathtt{kN} - 1]$

**end for**

**if** ($\mathtt{e} \mod 2 = 1$ ) **then**

  KernelTwiddleMultiplication($\vec{\mathtt{X}}, \vec{\Omega}, \mathtt{N}, \mathtt{K}, \mathtt{k}, \mathtt{s} := 2, \mathtt{r}$)

  HostGeneralStridePermutation($\vec{\mathtt{X}}, \vec{\mathtt{Y}}, \mathtt{K}^1, \mathtt{N}, \mathtt{k}, \mathtt{s}, \mathtt{b}$)

  $\vec{\mathtt{X}}[0 : \mathtt{kN} - 1] := \vec{\mathtt{Y}}[0 : \mathtt{kN} - 1]$

  KernelBaseDFTKAllSteps($\vec{\mathtt{X}}, \mathtt{N}, \mathtt{K}, \mathtt{r}$)

  HostGeneralStridePermutation($\vec{\mathtt{X}}, \vec{\mathtt{Y}}, \mathtt{K}^1, \mathtt{N}, \mathtt{k}, \mathtt{s}, \mathtt{b}$)

  $\vec{\mathtt{X}}[0 : \mathtt{kN} - 1] := \vec{\mathtt{Y}}[0 : \mathtt{kN} - 1]$

**end if**

**return** $\vec{\mathtt{X}}$

## 6.5 Profiling results

Our implementation is optimized for the prime $p = r^8 + 1$ with radix $r = 2^{63} + 2^{34}$. Figure 6.1 presents running-time diagram for computing $\mathrm{DFT}_{K^4}$ with $K = 16$ on a randomly generated vector over $\mathbb{Z}/p\mathbb{Z}$. We have collected the profiling data on a NVIDIA Geforce-GTX760M card (hardware specifications are mentioned in Appendix B).

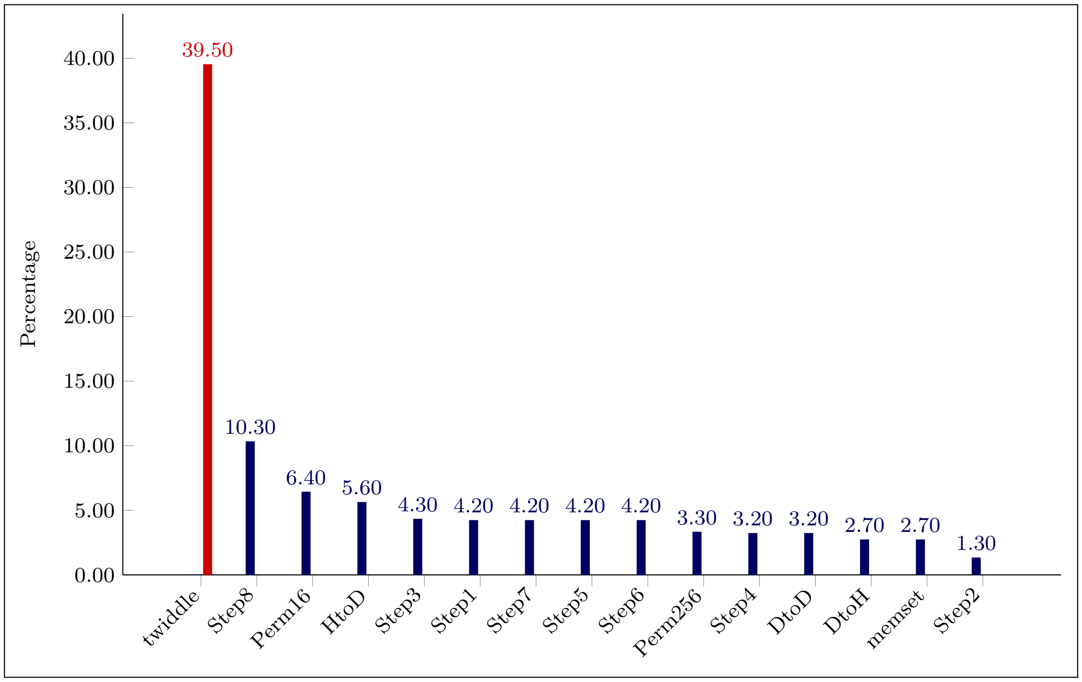


Figure 6.1: Running-time for computing $\mathrm{DFT}_N$ with $N = K^4$ and $K = 16$.

# Chapter 7

# Experimental Results: Big Prime Field FFT vs Small Prime Field FFT

In this chapter, we compare our implementation of FFT over a big prime field against one over a small prime field. Recall that a *big prime field* refers to a finite field of the form $\mathbb{Z}/p\mathbb{Z}$, where the binary representation of $p$ requires multiple machine-words. Meanwhile a *small prime field* refers to the case where the prime characteristic $p$ can be represented within a single machine-word. In Section 7.1, we explain how the *reverse mixed-radix conversion* [22] helps us have a fair comparison between our big prime field FFT and the small prime field FFT. Then, in Section 7.2, we develop two benchmarks for measuring the performance of big and small prime field FFTs. Finally, in Section 7.3, we report on the experimental results of those benchmarks. For the small prime field approach, we rely on the CUMODP library [15].

## 7.1 Background

Returning to the discussion of Chapter 1, we are interested in comparing the following approaches:

**Big prime:** For a big prime field $\mathbb{Z}/p\mathbb{Z}$, for a polynomial $f \in \mathbb{Z}/p\mathbb{Z}[x]$ of degree $N-1$, where $N$ is a power of 2 and $p = r^k + 1$ is a generalized Fermat prime, such that $N$ divides $p-1$ and $k$ is a power of 2, compute the DFT of $f$ at an $N$-th primitive root of unity $\omega$ such that $\omega^{N/2k} = r$ and $r$ is of machine-word size.

**Small primes:** For pairwise different prime numbers $p_1, \ldots, p_k$ of machine-word size, for a polynomial $f \in \mathbb{Z}/m\mathbb{Z}[x]$ of degree $N-1$, where $N$ is as above and divides each of $p_1 - 1, \ldots, p_k - 1$, compute the DFT of $f$ at $\omega = (\omega_1, \ldots, \omega_k)$ where $\omega_i$ is an $N$-th primitive root of unity in $\mathbb{Z}/p_i\mathbb{Z}$, for $i = 1, \ldots, k$, using the isomorphism $\mathbb{Z}/m\mathbb{Z}[x] \simeq \mathbb{Z}/p_1\mathbb{Z}[x] \oplus \cdots \oplus \mathbb{Z}/p_k\mathbb{Z}[x]$.

The first approach is what we have explained from Chapter 3 to Chapter 6. In the rest of this section, we discuss the second approach.

We recall from the introduction that the second approach can be done in three steps:

1. ***projection:*** compute the image $f_i$ of $f$ in $\mathbb{Z}/p_1\mathbb{Z}[x], \ldots, \mathbb{Z}/p_k\mathbb{Z}[x]$, for $i = 1, \ldots, k$,
2. ***images:*** compute the DFT of $f_i$ at $\omega_i$ in $\mathbb{Z}/p_i\mathbb{Z}[x]$, for $i = 1, \ldots, k$,
3. ***combination:*** combine the results using CRT so as to obtain a DFT of $f$ at $\omega$.

We observe that the first and third steps have similar algebraic costs, namely $O(k \times N \times k^2)$ machine-word operations and similar memory access patterns. For this reason, our implementation is based on the second and third steps only. This is sufficient to have a practical estimate of the overall cost of the small prime field approach. Implementing the first step is work in progress.

The second step is realized with the code available in the CUMODP library, developed as part of the work reported in [17] by Wei Pan and Marc Moreno Maza. Hence, from now on, we focus on the above third step, that is, the *recombination*. We observe that we need to combine $k$ vectors component-wise, namely the DFTs of $f_1, \ldots, f_k$, into a single vector, namely the DFT of $f$. However, the prime numbers used in [17] are of half a machine-word size. Thus, we should use $2k$ primes in our small prime field approach in order to obtain a fair comparison with the big prime field approach. For this reason, in the sequel, we discuss the recombination of $s$ vectors, where $s = k$ in theory and $s = 2k$ in practice.

So let $b_1, \ldots, b_s$ be elements of $\mathbb{Z}/p_1\mathbb{Z}, \ldots, \mathbb{Z}/p_s\mathbb{Z}$, where $p_1, \ldots, p_s$ are pairwise different prime numbers of machine-word size, or less. We assume that the elements of $\mathbb{Z}/p_1\mathbb{Z}, \ldots, \mathbb{Z}/p_s\mathbb{Z}$ are encoded with a *non-negative representation*, thus, we have $0 \leq b_i < p_i$ for all $i = 1, \ldots, s$. Then $(b_1, b_2, \ldots, b_{(s)})$ is the *mixed-radix representation* of the integer $n \in \mathbb{Z}$ given by

$$n = b_1 + b_2 p_1 + b_3 p_1 p_2 + \cdots + b_s p_1 \cdots p_{s-1}. \tag{7.1}$$

Note that we have

$$0 \leq n < p_1 p_2 \cdots p_s. \tag{7.2}$$

In our recombination step, we use the mixed radix representation map (given by Formula (7.1)) rather than the CRT. The latter defines a ring isomorphism between $\mathbb{Z}/p_1\mathbb{Z} \oplus \cdots \oplus \mathbb{Z}/p_s\mathbb{Z}$ and $\mathbb{Z}/m\mathbb{Z}$ where $m$ is the product of the primes $p_1, \ldots, p_s$. Meanwhile, the former defines a bijection between $\mathbb{Z}/p_1\mathbb{Z} \oplus \cdots \oplus \mathbb{Z}/p_s\mathbb{Z}$ and the integer range $[0, p_1 p_2 \cdots p_s]$ with the property that integers in that range can be compared (in terms of the natural total order $<$) simply by comparing lexicographically their mixed-radix representations. For modular methods dealing with real numbers, say for real root isolation of univariate polynomials, the mixed radix representation is of great interest, see [5]. This is why, we chose mixed radix representation in our recombination step. Note that both recombination schemes have similar algebraic complexity, namely $\Theta(k^2)$ machine-word operations.

After pre-computing the products $m_1 := p_1$, $m_2 := p_1 p_2$, $\ldots$, $m_s := m$, we reconstruct any integer $n$ from its mixed-radix representation $(b_1, b_2, \ldots, b_s)$ using Formula (7.1) as follows:

1. each product $u_i := b_i m_i$ will be computed and stored in $i$ machine-words, for $i = 1, \ldots, s$, then
2. the sum $n := u_1 + u_2 + \ldots + u_s$ will be computed as the final result of the conversion.

In our GPU implementation, the precomputed products $m_1, \ldots, m_s$ will be stored in global memory. Also, exactly one thread will be assigned for computing each conversion from a mixed-radix representation $(b_1, b_2, \ldots, b_s)$ to the corresponding integer $n$. Recall that, in our implementation $p = r^8 + 1$, with $r = 2^{63} + 2^{34}$ and we have $s = 2k = 16$. The CUDA source-code shown in Appendix C.1, with precomputed products using the first 16 primes in Table A.1.

## 7.2 Comparing FFT over small and big prime fields

In this section, we explain how we compare the performance of FFT computation over a big prime field against that of FFT computation over a small prime field. We develop two benchmarks:

1. one comparing the running-time when the two computations produce similar result, and thus the same amount of output data,
2. one comparing the running-time when the two coomputations process the same amount of input data.

### 7.2.1 Benchmark 1: Comparison when computations produce the same amount of output data

This first benchmark corresponds to the comparison described in Section 7.1. We observe that the output of the two approaches is a the DFT of a vector of size $N$ over a direct product $R$ of prime fields where each element of $R$ spans $k$ machine-words. Hence these two approaches can be equivalent building blocks in a modular method. We observe that the small prime field approach

1. performs more memory traversal (due to the projection and combination steps) and clearly has a higher cache complexity, meanwhile,
2. the same small prime field approach has a lower algebraic complexity as explained in Chapter 1.

### 7.2.2 Benchmark 2: Comparison when computations process the same amount of input data

Since memory access patterns play an essential role in the performance of FFT computations, it is natural to try to compare the big prime field and small prime field calculations in a situations where they process the same amount of data. So, instead of computing equivalent results as in Benchmark 1, we simply ensure that the same amount of data is read. To do so, we simply change the small prime field calculations as follows: we perform $s$ FFTs of size $N$ over small prime fields, where $s = 2k$ and small primes are of half of a machine-word in size. Therefore, in both calculations, the same amount of data is processed, namely $k\,N$ machine words.

We stress the fact that the results produced by the two approaches are not algebraically equivalent. The intention is to check whether the big prime field calculation has similar performance than another (and actually highly optimized) FFT calculation processing the same amount of data.

## 7.3 Benchmark results

We use the following algorithms from the CUMODP library for computing the FFT over a small prime field:

1. the Cooley-Tukey FFT algorithm with precomputed powers of the primitive root,
2. the Cooley-Tukey FFT algorithm without precomputation, and

3. the Stockham FFT algorithm.

See [17] for details. Due to the size of the global memory on a GPU card, the above algorithms can compute DFTs for input vectors of $2^n$ elements, where $n \leq 26$ is typical. For implementation design reasons, we also have $8 \leq n$. Note that these FFT implementations use 32-bit Fourier primes from Table A.1.

We observe that the small prime field FFT codes of the CUMODP library are highly optimized: they have been continuously improved since their initial release [17, 18] until very recently [15]. The experimental results in those papers show that CUMODP's small prime field FFT codes outperform serial small prime field FFT codes by large factors, typically 30 to 40 on a Tesla 2050 NVIDA GPU card.

For computing DFT over $\mathbb{Z}/p\mathbb{Z}$, with $p = (2^{63} + 2^{34})^8 + 1$, we use our CUDA implementation of the algorithms presented in Chapter 6. The size of the input vectors is $N = K^e$, with $K = 16$ and $2 \leq e \leq 5$.

Benchmarks are measured on a NVIDA GeforceGTX760M card (hardware specifications are mentioned in Appendix B). Figures 7.1 (Benchmark 1) and 7.2 (Benchmark 2) show running-time ratios between the three small prime field FFTs and the big prime field FFT, for the following 4 values of $N$, namely $N = K^2$, $N = K^3$, $N = K^2$ and $N = K^4$. Also, Tables 7.1, 7.2, 7.3, 7.4, and 7.5 present running time (in milliseconds) of computing Benchmark 1 and Benchmark 2.

### 7.3.1 Performance analysis.

As it is reported in [17], FFT algorithms of the CUMODP library gain speed-up factors for vectors of the size $2^{16}$ and larger. In other words, the input vector should be large enough to keep the GPU device busy, and therefore, provide a high percentage of occupancy. This explains the results displayed on Figure 7.1 (Benchmark 1) and 7.2 (Benchmark 2) for $N = K^2$ and $N = K^3$, that is, why apparently the big prime field FFT approach seems to outperform the small prime field FFT approach.

For $N = K^5$, the Cooley-Tukey (with precomputation) and Stockham FFT codes are essentially twice faster than the big prime field FFT (see Benchmark 1). For $N = K^4$, only the Cooley-Tukey (with precomputation) outperforms the big prime field FFT (see Benchmark 1) and this is only by a 10% factor.

We view this as a promising result for the big prime field FFT since

1. the small prime field FFT codes have been developed and optimized for more than 8 years,
2. the projection part of the small prime field FFT approach is not implemented yet which is unfair to the big prime field FFT approach,
3. the small prime field FFT codes rely mostly on 32-bit arithmetic meanwhile the the big prime field FFT code is implemented in 64-bit arithmetic, for which CUDA provides less opportunities for optimization such as instruction level parallelism[1].

| Computation | CT-precomp FFT | CT FFT | Stockham FFT | Big FFT |
|---|---|---|---|---|
| 16 FFTs | 1.944 (ms) | 7.330 (ms) | 4.672 (ms) | 0.038 (ms) |
| 16 FFTs + M.R.C. | 2.192 (ms) | 7.595 (ms) | 4.824 (ms) | 0.038 (ms) |

Table 7.1: Running time of computing Benchmark 1 for $N = K^2$ with $K = 16$.

| Computation | CT-precomp FFT | CT FFT | Stockham FFT | Big FFT |
|---|---|---|---|---|
| 16 FFTs | 5.061 (ms) | 9.912 (ms) | 7.491 (ms) | 0.384 (ms) |
| 16 FFTs + M.R.C. | 5.399 (ms) | 9.855 (ms) | 7.266 (ms) | 0.384 (ms) |

Table 7.2: Running time of computing Benchmark 1 for $N = K^3$ with $K = 16$.

| Computation | CT-precomp FFT | CT FFT | Stockham FFT | Big FFT |
|---|---|---|---|---|
| 16 FFTs | 13.764 (ms) | 35.952 (ms) | 19.001 (ms) | 22.414 (ms) |
| 16 FFTs + M.R.C. | 19.892 (ms) | 40.176 (ms) | 24.426 (ms) | 22.414 (ms) |

Table 7.3: Running time of computing Benchmark 1 for $N = K^4$ with $K = 16$.

| Computation | CT-precomp FFT | CT FFT | Stockham FFT | Big FFT |
|---|---|---|---|---|
| 16 FFTs | 158.159 (ms) | 564.290 (ms) | 222.554 (ms) | 468.464 (ms) |
| 16 FFTs + M.R.C. | 250.736 (ms) | 648.196 (ms) | 287.315 (ms) | 468.464 (ms) |

Table 7.4: Running time of computing Benchmark 1 for $N = K^5$ with $K = 16$.

| e | CT-precomp FFT | CT FFT | Stockham FFT | Big FFT |
|---|---|---|---|---|
| 2 | 0.329 (ms) | 0.609 (ms) | 0.453 (ms) | 0.038 (ms) |
| 3 | 0.841 (ms) | 2.130 (ms) | 1.147 (ms) | 0.384 (ms) |
| 4 | 9.874 (ms) | 34.971 (ms) | 11.956 (ms) | 22.414 (ms) |
| 5 | 170.624 (ms) | 736.450 (ms) | 215.869 (ms) | 468.464 (ms) |

Table 7.5: Running time of computing Benchmark 2 for $N = K^e$ with $K = 16$.

[1] https://en.wikipedia.org/wiki/Instruction-level_parallelism

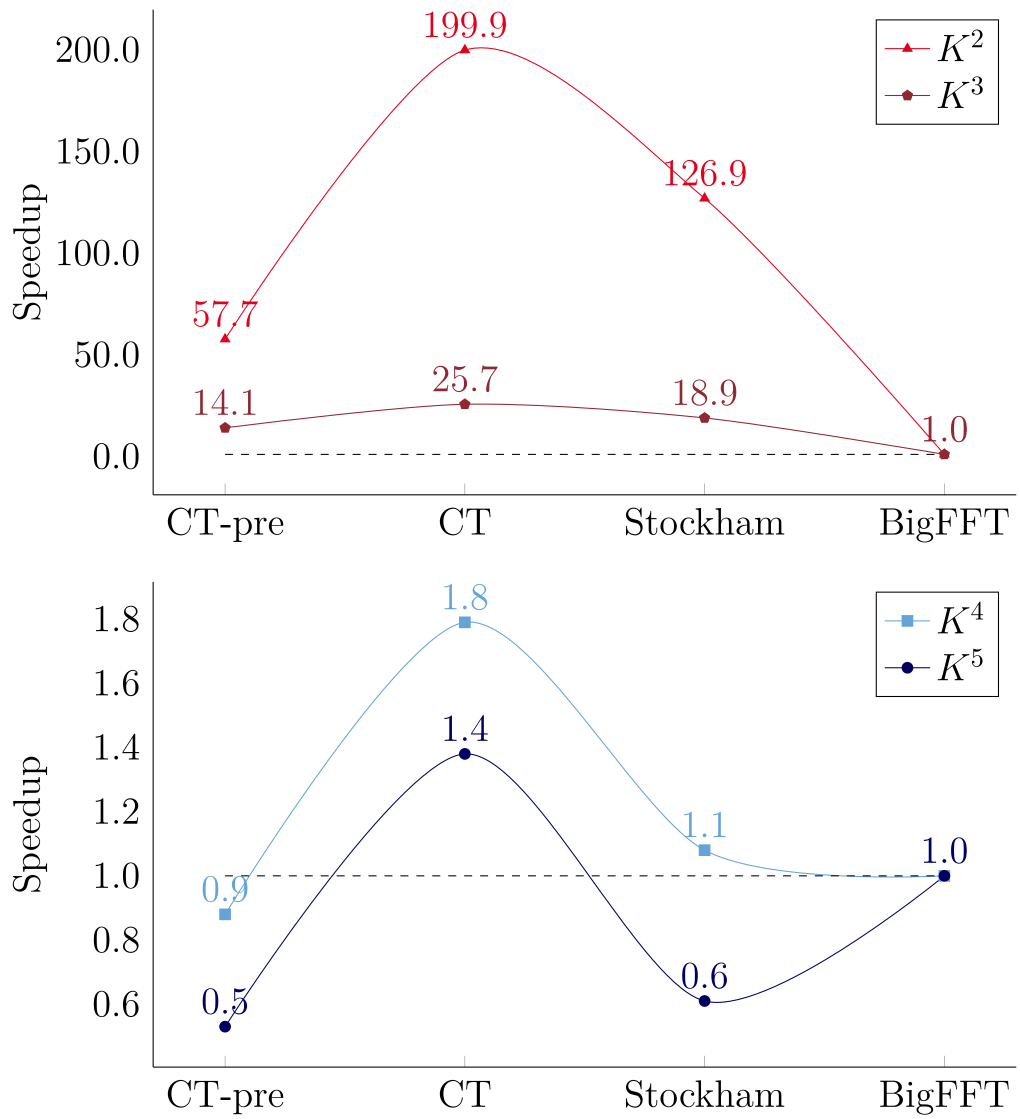


Figure 7.1: Speed-up diagram of Benchmark 1 for $K = 16$.

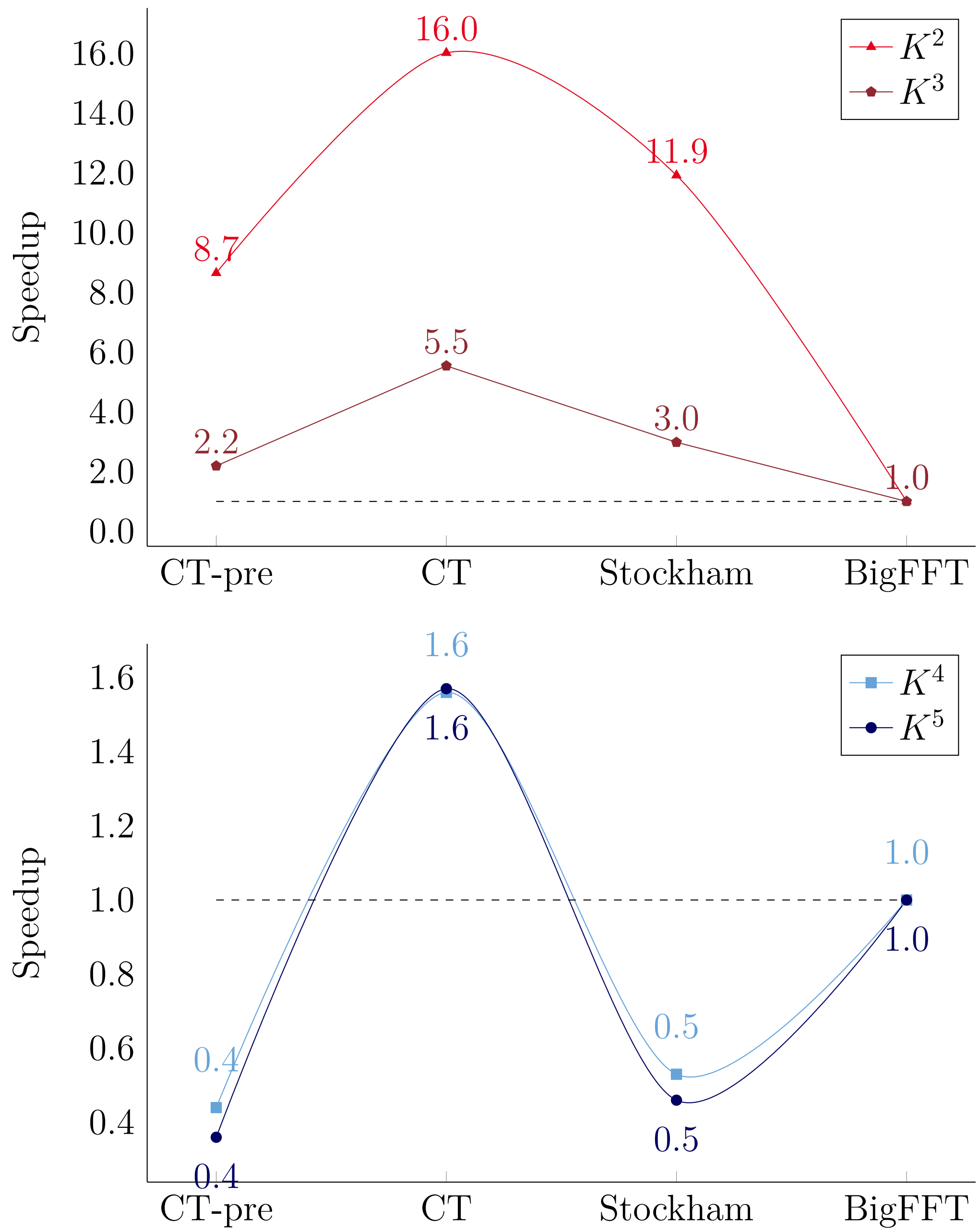


Figure 7.2: Speed-up diagram of Benchmark 2 for $K = 16$.

## 7.4 Concluding remarks

As discussed in Chapter 1, big prime field arithmetic is required by advanced algorithms in computer algebra (like polynomial system solving). As demonstrated in Chapter 4, arithmetic modulo a big prime can be efficiently computed on GPUs in the case of Generalized Fermat primes. Nevertheless, multiplication in $\mathbb{Z}/p\mathbb{Z}$ (except for the case of a multiplication by a power of $r$) remains a computational bottleneck, as illustrated in the same Chapter 4. Improving multiplication in $\mathbb{Z}/p\mathbb{Z}$ is work in progress. Moreover, by choosing larger primes, say with $k = 16$ instead of $k = 8$, we hope to cover other ranges for the vectors to which big prime field FFT is applied.

# Bibliography


[1] E. A. Arnold. Modular algorithms for computing Gröbner bases. *J. Symb. Comput.*, 35(4):403–419, April 2003.

[2] C. Chen, R. M. Corless, M. Moreno Maza, P. Yu, and Y. Zhang. An application of regular chain theory to the study of limit cycles. *I. J. Bifurcation and Chaos*, 23(9), 2013.

[3] C. Chen, S. Covanov, F. Mansouri, M. Moreno Maza, N. Xie, and Y. Xie. The basic polynomial algebra subprograms. In H. Hong and C. Yap, editors, *Mathematical Software - ICMS 2014 - 4th International Congress, Seoul, South Korea, August 5-9, 2014. Proceedings*, volume 8592 of *Lecture Notes in Computer Science*, pages 669–676. Springer, 2014.

[4] C. Chen, S. Covanov, F. Mansouri, M. Moreno Maza, N. Xie, and Y. Xie. Parallel integer polynomial multiplication. In *International Symposium on Symbolic and Numeric Algorithms for Scientific Computing*. IEEE Society, 2016. To appear.

[5] C. Chen, M. Moreno Maza, and Y. Xie. Cache complexity and multicore implementation for univariate real root isolation. *J. of Physics: Conference Series*, 341, 2011.

[6] J. Cooley and J. Tukey. An algorithm for the machine calculation of complex fourier series. *Mathematics of Computation*, 19(90):297–301, 1965.

[7] NVIDIA Corporation. CUDA C Best Practices Guide, v8.0, September 2016.

[8] NVIDIA Corporation. CUDA C Programming Guide, v8.0, September 2016.

[9] X. Dahan, M. Moreno Maza, É. Schost, W. Wu, and Y. Xie. Lifting techniques for triangular decompositions. In M. Kauers, editor, *Symbolic and Algebraic Com-*

*putation, International Symposium ISSAC 2005, Beijing, China, July 24-27, 2005, Proceedings*, pages 108–115. ACM, 2005.

[10] F. Franchetti and M. Püschel. FFT (fast fourier transform). In D. A. Padua, editor, *Encyclopedia of Parallel Computing*, pages 658–671. Springer, 2011.

[11] M. Frigo and S. G. Johnson. The design and implementation of FFTW3. In *Proceedings of the IEEE*, volume 93, pages 216–231, 2005.

[12] M. Fürer. Faster integer multiplication. In D. S. Johnson and U. Feige, editors, *Proceedings of the 39th Annual ACM Symposium on Theory of Computing, San Diego, California, USA, June 11-13, 2007*, pages 57–66. ACM, 2007.

[13] M. Fürer. Faster integer multiplication. *SIAM J. Comput.*, 39(3):979–1005, 2009.

[14] D. Hankerson, A. J. Menezes, and S. Vanstone. *Guide to Elliptic Curve Cryptography*. Springer-Verlag New York, Inc., 2004.

[15] S. A. Haque, X. Li, F. Mansouri, M. Moreno Maza, W. Pan, and N. Xie. Dense arithmetic over finite fields with the CUMODP library. In H. Hong and C. Yap, editors, *Mathematical Software - ICMS 2014 - 4th International Congress, Seoul, South Korea, August 5-9, 2014. Proceedings*, volume 8592 of *Lecture Notes in Computer Science*, pages 725–732. Springer, 2014.

[16] X. Li, M. Moreno Maza, R. Rasheed, and É. Schost. The modpn library: Bringing fast polynomial arithmetic into maple. *J. Symb. Comput.*, 46(7):841–858, 2011.

[17] M. Moreno Maza and W. Pan. Fast polynomial arithmetic on a GPU. *J. of Physics: Conference Series*, 256, 2010.

[18] M. Moreno Maza and W. Pan. Solving bivariate polynomial systems on a GPU. *J. of Physics: Conference Series*, 341, 2011.

[19] J. Nickolls, I. Buck, M. Garland, and K. Skadron. Scalable parallel programming with CUDA. *ACM Queue*, 6(2):40–53, 2008.

[20] M. Püschel, J. M. F. Moura, J. Johnson, D. Padua, M. Veloso, B. Singer, J. Xiong, F. Franchetti, A. Gacic, Y. Voronenko, K. Chen, R. W. Johnson, and N. Rizzolo. SPIRAL: Code generation for DSP transforms. *Proceedings of the IEEE, special issue on "Program Generation, Optimization, and Adaptation"*, 93(2):232– 275, 2005.

[21] A. Schönhage and V. Strassen. Schnelle multiplikation großer zahlen. *Computing*, 7(3-4):281–292, 1971.

[22] J. Schönheim. Conversion of modular numbers to their mixed radix representation by matrix formula. *Mathematics of Computation*, 21(98):253–257, 1967.

[23] J. von zur Gathen and J. Gerhard. Fast algorithms for taylor shifts and certain difference equations. In *ISSAC*, pages 40–47, 1997.

[24] J. von zur Gathen and J. Gerhard. *Modern Computer Algebra*. Cambridge University Press, New York, NY, USA, 2 edition, 2003.

# Appendix A

# Table of 32-bit Fourier primes

| | | | | | |
|---|---|---|---|---|---|
| 962592769 | 957349889 | 950009857 | 943718401 | 940572673 | 938475521 |
| 935329793 | 925892609 | 924844033 | 919601153 | 918552577 | 913309697 |
| 907018241 | 899678209 | 897581057 | 883949569 | 880803841 | 862978049 |
| 850395137 | 833617921 | 824180737 | 802160641 | 800063489 | 818937857 |
| 799014913 | 786432001 | 770703361 | 754974721 | 745537537 | 740294657 |
| 718274561 | 715128833 | 710934529 | 683671553 | 666894337 | 655360001 |
| 648019969 | 645922817 | 639631361 | 635437057 | 605028353 | 597688321 |
| 595591169 | 581959681 | 576716801 | 531628033 | 493879297 | 469762049 |
| 468713473 | 463470593 | 459276289 | 447741953 | 415236097 | 409993217 |
| 399507457 | 387973121 | 383778817 | 377487361 | 361758721 | 359661569 |
| 347078657 | 330301441 | 311427073 | 305135617 | 290455553 | 274726913 |
| 270532609 | 257949697 | 249561089 | 246415361 | 230686721 | 221249537 |
| 211812353 | 204472321 | 199229441 | 186646529 | 185597953 | 169869313 |
| 167772161 | 163577857 | 158334977 | 155189249 | 147849217 | 141557761 |
| 138412033 | 136314881 | 132120577 | 120586241 | 113246209 | 111149057 |
| 104857601 | 101711873 | 81788929 | 70254593 | 69206017 | 28311553 |

Table A.1: Table of 32-bit Fourier primes.

# Appendix B

# Hardware specification

## B.1 GeforceGTX760M (Kepler)

```
Device 0: "GeForce GTX 760M"
  CUDA Capability Major/Minor version number:    3.0
  Total amount of global memory:                 2048 MBytes (2147352576 bytes)
  ( 4) Multiprocessors, (192) CUDA Cores/MP:     768 CUDA Cores
  GPU Max Clock rate:                            719 MHz (0.72 GHz)
  Memory Clock rate:                             2004 Mhz
  Memory Bus Width:                              128-bit
  L2 Cache Size:                                 262144 bytes
  Maximum Layered 1D Texture Size, (num) layers  1D=(16384), 2048 layers
  Maximum Layered 2D Texture Size, (num) layers  2D=(16384, 16384), 2048 layers
  Total amount of constant memory:               65536 bytes
  Total amount of shared memory per block:       49152 bytes
  Total number of registers available per block: 65536
  Warp size:                                     32
```

Figure B.1: Hardware specification for NVIDIA GeforceGTX760M.

**Theoretical bandwidth.** We compute the theoretical memory bandwidth of this device based on data presented in Figure B.1, and by using the equation that explained in Chapter 2:

$$\begin{aligned} B_T &:= 2.004 \times 10^9 \times 128/8 \times 2, \\ B_T &:= 64.12 \text{ GB/s}. \end{aligned}$$

**Practical bandwidth.** As it is presented in Figure B.2, the value of effective bandwidth (in this case, bandwidth of device to device transfer) is 48.8 GB/s, which is equal to almost 70% of the theoretical bandwidth.

```
Running on...
 Device 0: GeForce GTX 760M
 Device to Device Bandwidth, 1 Device(s)
 PINNED Memory Transfers
   Transfer Size (Bytes) Bandwidth(MB/s)
   33554432   48820.3
```

Figure B.2: The bandwidth test from CUDA SDK (`samples/1_Utilites/bandwidthTest`).

# Appendix C

# Source code

## C.1 Kernel for computing reverse mixed-radix conversion

```
typedef unsigned int usfixn32;
typedef unsigned long long int usfixn64;

__device__ void device_sum_17_u32(usfixn32 * s, usfixn32 *r, usfixn32 step)
{
    usfixn32 i=0, sum=0,carry=0;
    for(i=0;i<step;i++)
    {
     sum=s[i]+r[i];
     if(sum<s[i]||sum<r[i])
             r[i+1]++;
     r[i]=sum;
     s[i]=0;
    }
}
__global__ void kernel_crt_multiplications_v1(vs, precomputePrimes,
  result, parameters)
{
 usfixn32 tid = threadIdx.x + blockIdx.x * blockDim.x;
```

```
usfixn32 nPrimes = 16;
usfixn32 i=0, j=0, c=0, k=0;
usfixn32 r[17]={0};
j = threadIdx.x & (0xF);
usfixn64 mult=0,sum = 0,offset = 0;
usfixn32 permutationStride = parameters[5];
usfixn32 n = parameters[0];
usfixn64 tmp;
if (tid >= n)
 return;
usfixn32 m0;
usfixn64 carry = 0;
if (j > 0)
 for (i = 0; i < nPrimes; i++)
  m[i]=precomputePrimes[j][i];

for (j = 0; j < 16; j++)
{
 tmp = vs[tid + j * permutationStride];
 if (j == 0)
 {
  s[0] = tmp;
 }
 carry = 0;
 m0 = 0;
 if (j > 0)
 {
  for (i = 0; i < j + 1; i++)
  {
   mult = usfixn64(tmp * precomputePrimes[j-1][i]);
   m0 = (mult & 0xFFFFFFFF);
   sum = s[i] + m0 + carry;
   s[i] = (sum & 0xFFFFFFFF);
   mult >>= 32;
   sum >>= 32;
   carry = (mult) + sum;
  }
 }
```

```
  device_sum_17_u32(s,r,j);
 }
 offset=0;
 for (i = 0; i < nPrimes; i++)
 {
  result[tid + offset] = r[i];
  offset += permutationStride;
 }
}
```

# Curriculum Vitae

| | |
|---|---|
| **Name:** | **Davood Mohajerani** |
| **Post-Secondary Education and Degrees:** | University of Western Ontario<br>London, Ontario, Canada<br>M.Sc. in Computer Science, 2015 - 2016 |
| | Isfahan University of Technology<br>Isfahan, Iran<br>B.Sc. in Computer Engineering, 2010 - 2015 |
| **Related Work Experience:** | Research Assistant/Teaching Assistant<br>University of Western Ontario<br>2015 - 2016 |