Fast Multiplication in Binary Fields on GPUs via Register Cache

Eli Ben-Sasson
Technion
Haifa, Israel
eli@cs.technion.ac.il

Matan Hamilis
Technion
Haifa, Israel
hamilis@cs.technion.ac.il

Mark Silberstein
Technion
Haifa, Israel
mark@ee.technion.ac.il

Eran Tromer
Tel-Aviv University
Tel-Aviv, Israel
tromer@cs.tau.ac.il

ABSTRACT

Finite fields of characteristic 2 – "binary fields" – are used in a variety of applications in cryptography and data storage. Multiplication of two finite field elements is a fundamental operation and a well-known computational bottleneck in many of these applications, as they often require multiplication of a large number of elements. In this work we focus on accelerating multiplication in "large" binary fields of sizes greater than 2^{32}. We devise a new parallel algorithm optimized for execution on GPUs. This algorithm makes it possible to multiply large number of finite field elements, and achieves high performance via *bit-slicing* and fine-grained parallelization.

The key to the efficient implementation of the algorithm is a novel performance optimization methodology we call the *register cache*. This methodology speeds up an algorithm that caches its input in shared memory by transforming the code to use per-thread registers instead. We show how to replace shared memory accesses with the `shuffle()` intra-warp communication instruction, thereby significantly reducing or even eliminating shared memory accesses. We thoroughly analyze the register cache approach and characterize its benefits and limitations.

We apply the register cache methodology to the implementation of the binary finite field multiplication algorithm on GPUs. We achieve up to $138\times$ speedup for fields of size 2^{32} over the popular, highly optimized Number Theory Library (NTL) [26], which uses the specialized `CLMUL` CPU instruction, and over $30\times$ for larger fields of size below 2^{256}. Our register cache implementation enables up to 50% higher performance compared to the traditional shared-memory based design.

ICS '16 June 1–3, 2016, Istanbul, Turkey

DOI: 10.475/123_4

CCS Concepts

•Computer systems organization → Single instruction, multiple data; •Mathematics of computing → *Mathematical software performance;*

Keywords

GPGPU; SIMD; Finite Field Multiplication; Parallel Algorithms;GPU Code Optimization

1. INTRODUCTION

A *binary* field is a Galois field (GF) (also called finite field) of size 2^n, where n is an integer; we denote it by $\mathrm{GF}(2^n)$. Binary fields have numerous applications in cryptography and data storage. For instance, the Advanced Encryption Standard (AES) [13] uses $\mathrm{GF}(2^8)$, as does the error correction scheme used on Compact Discs (CDs) and Digital Versatile Discs (DVDs). Large fields are the basis for distributed storage systems like those used by Google and Amazon, which employ fields of size $2^{32}, 2^{64}$ and 2^{128} to ensure secure and reliable storage of data on multiple disks [15]. They are also the basis for the application motivating this work: an efficient implementation of a family of probabilistically checkable proofs (PCP) of quasi-linear length [8]. This application is envisioned to enable verifiable execution, whereby a client that offloads a computation to untrusted computing resources, e.g., to a cloud system, receives a proof which attests that the results have indeed been produced by the execution of the offloaded computation. PCPs require very large binary fields: most of our work focuses on $\mathrm{GF}(2^{32})$ and $\mathrm{GF}(2^{64})$ but we also support fields of up to $\mathrm{GF}(2^{2048})$. Because all the applications mentioned above need to perform multiplication of a large

number of finite field elements, their performance is dominated by the cost of finite field multiplication, motivating the never-ending quest for more efficient implementations of this fundamental arithmetic operation.

In this paper we focus on accelerating finite field multiplication for large binary extension fields of size larger than $\mathrm{GF}(2^{32})$ on GPUs, where field elements are represented using a standard basis (cf. Section 4 for definitions). The main computational bottleneck in this case is the multiplication of polynomials over $\mathrm{GF}(2)$, that is, polynomials with $\{0,1\}$-coefficients. The challenge posed by polynomial multiplication operations over $\mathrm{GF}(2)$ has led Intel and AMD to add an instruction set extension `CLMUL` to support it in hardware.

We devise a novel parallel algorithm for multiplication in large binary extension fields on GPUs, which significantly outperforms the dedicated CPU hardware implementation. The algorithm is based on two main ideas: First, we apply *bit-slicing*, enabling a single thread to perform *32 multiplications in parallel*. As a result, all the arithmetic operations involved in multiplication are performed on 32 bits together instead of a single bit at a time for single multiplication, therefore matching the width of hardware registers and enabling full ALU utilization. Second, the computation of a single multiplication is further parallelized in a fine-grained manner to eliminate execution divergence among the participating threads. This critical step allows these computations to be mapped to the threads of a single GPU warp, whose threads are executed in lock-step.

We then focus on an implementation of the algorithm on modern NVIDIA GPUs. The key to implementation efficiency is a novel optimization technique that we call the *register cache*. The register cache enables us to use per-thread registers in conjunction with the `shuffle()` intrinsics to construct a *register-based cache for threads in a single warp*. This cache serves the same purpose as the on-die shared memory, but is much faster thanks to higher bandwidth and reduced synchronization overhead. We propose a general methodology for transforming a traditional algorithm that stores its inputs in shared memory into a potentially more efficient one that uses private per-thread registers to cache the input for the warp's threads. We thoroughly study the benefits and limitations of the register cache approach on the example of a well-known k-stencil kernel.

Finally, we apply the register cache methodology to optimize the implementation of the finite field multiplication algorithm for $\mathrm{GF}(2^N)$, where N=32,...,2048. The primary challenge is to scale the efficient single-warp implementation to larger fields while retaining the performance benefits of the register cache methodology. We analyze several design options, and apply an algorithm that uses a low-degree multiplication as a building block for multiplication in larger fields.

We evaluate our implementation across a variety of field and input sizes using NVIDIA Titan-X GPU with 12GB of memory, and compare it to a highly optimized CPU version of a popular Number Theory Library (NTL) [26] running on a single core of Intel® Xeon® CPU E5-2620 v2 @ 2.10GHz that uses the Intel's CLMUL CPU instruction set extension. Our optimized implementation that uses register cache is up to $138\times$ faster than NTL for $\mathrm{GF}(2^{32})$ when multiplying more than 2^{25} finite field elements. The register cache approach enables us to speed up the original shared memory version by about 50% over all field sizes.

Our contributions in this paper are as follows:

1. A novel algorithm for polynomial multiplication over $\mathrm{GF}(2)$ on GPUs,
2. A general optimization methodology for using GPU registers as an intra-warp user-managed cache, along with an in depth analysis of this approach and its application to polynomial multiplication.
3. Efficient GPU finite field multiplication that is up to two orders of magnitude faster in fields ($\mathrm{GF}(2^{32})$) than the CPU implementation that uses the specialized hardware instruction.

This paper is organized as follows. In Sections 2–3 we introduce and analyze the register cache optimization approach, and highlight its benefits by applying it to a simple k-stencil kernel. In Sections 4–7 we explain the parallel algorithm for finite field multiplication on GPUs, and show how to apply the register cache methodology to boost its performance. We conclude with an evaluation (Section 9) and related work (Section 10).

2. GPU BACKGROUND

We briefly outline the basic concepts of the GPU architecture and programming model, using NVIDIA CUDA® terminology; more details about CUDA® and the GPU model can be found in [6].

Execution hierarchy. GPUs are parallel processors that run thousands of threads. Threads are grouped into warps, warps are grouped into threadblocks, and multiple threadblocks form a GPU kernel which is invoked on the GPU by the CPU. A warp is a set of 32 threads that are executed in lockstep, i.e., at a given step all the threads in the warp execute the same instruction. Threads in a threadblock are invoked on the same core, called the Streaming Multiprocessor (SM), and may communicate and synchronize efficiently.

Memory hierarchy. Each thread has a set of dedicated private registers. Registers are the fastest type of memory. Threads in the same threadblock share a fast, threadblock-private, on-die scratchpad called *shared* memory. The performance characteristics of shared memory are similar to those of an L1 cache. In particular, it has lower bandwidth than the registers [28]. Shared memory is implemented using 32 memory banks, such that accesses to each bank are served independently. Each bank holds a stride of memory addresses such that concurrent accesses to 32 consecutive 4-byte words are served by different banks concurrently. Concurrent accesses to the same bank are serialized, resulting in *bank conflict*. All threads in the kernel share *global* GPU memory. Global memory provides about an order of magnitude lower bandwidth than shared memory. Accesses to global memory are cached in a two-level hardware cache.

Inter-thread communication. Threads in the kernel may communicate via global memory. Threads in the same thread-

block may also communicate via shared memory, and synchronize by using efficient hardware barriers. Threads in the same warp can communicate via `shuffle(x, y)` intrinsics, exchanging values stored in their private registers without using shared memory as an intermediary.

The `shuffle(t_i, r)` instruction is executed concurrently by multiple threads in a single warp. It allows thread t_j to share a value r it stores in a register and directly read private data shared by thread t_i. Variable r holds the data shared by the caller with the other threads in the warp. The instruction returns the data that the caller thread reads from thread t_i. The results are well defined only if the target thread t_i also calls `shuffle()`. The instruction supports arbitrary communication patterns between the warp threads, but allows a given thread to share only one 4-byte value at a time.

Thread divergence. Threads in a warp are executed in lockstep; therefore they are best suited to a *single instruction – multiple data* (SIMD) execution pattern. However, from the programming perspective, each thread in a warp is independent and may run its own code. When the warp threads encounter different, *divergent*, execution paths their execution is serialized. Therefore, to optimize performance, programmers seek algorithms that minimize or completely avoid divergence among threads.

Throughout the paper we denote by $\mathcal{W}$ the number of threads in a warp, by $\mathcal{TB}$ the number of threads in a threadblock and $\mathcal{B}$ as the width of hardware registers in bits. In current GPUs $\mathcal{W} = 32$ and $\mathcal{B} = 32$.

3. INTRA-WARP REGISTER CACHE

On-die shared memory is commonly used for data sharing among threads in the same threadblock. One common practice is to optimize input data reuse, whereby the kernel input is first prefetched from the global memory into the shared memory, thus saving global memory access cost on the following accesses.

In this section we focus on the *register cache*, a design methodology whose goal is to improve kernel performance by transforming computations to use registers instead of shared memory. We use private registers in each thread as a distributed storage, effectively implementing a layer of user-managed cache for the threads in the same warp with the help of the `shuffle()` instruction.

The benefits of using registers and `shuffle()` are well known in SIMD architectures [4], and are embraced in GPU computing [20, 27, 21, 1, 23, 11]. The `shuffle()`-based design removes the threadblock-wise synchronization overhead associated with the use of shared memory, and allows higher effective memory bandwidth to the data stored in registers. However, the existing uses of `shuffle()` are application-specific and offer no guidance for the design of the algorithm. Here we suggest a systematic approach to constructing `shuffle()`-based algorithms, aiming specifically to optimize applications with significant input data reuse.

Problem setting. We consider a common application scenario in which threads prefetch the shared threadblock input into shared memory and then access it repeatedly. Our goal is to reduce the use of shared memory as much as possible by identifying sharing and access patterns among the threads of a single warp, and replacing certain or all shared memory accesses by `shuffle()`.

Overview. We start with a shared memory-based implementation. The following steps accomplish the kernel transformation to use registers instead.

1. Identify warp inputs in shared memory.
2. Distribute inputs across warp threads such that each thread stores some part of the shared input in its registers. The optimal distribution is application dependent.
3. Logically break computations into two interleaving bulk-synchronous phases: communication and computation. The communication phase corresponds to the shared memory accesses in the original implementation. The computation phase is similar to the original implementation, but uses only the data in local registers.

Communication phase. We now describe the communication phase transformations in greater detail.

1. For each thread, declare the data to be read from other warp threads. We refer to each access as a `Read`(var, tid) operation, such that tid is the thread to read from, and var is the remote variable holding the data, both determined by the data distribution.
2. For each thread, compile the list of local variables required for the other threads by observing `Read` operations issued by them. Declare each such variable using `Publish`(var) operations.
3. Align `Read` and `Publish` operations in each thread and across the threads, such that (a) there is one `Read` for each `Publish` in each thread, and (b) there is one `Publish` for the value in the remote thread for each local `Read`. This step might require duplicating some calls to achieve perfect alignment, and/or redistribution of the inputs to reduce *conflicts*, i.e., when aligned `Read` requests from different threads need different variables from the same thread. Replace `Read-Publish` tuples with `shuffle()` calls.

3.1 Example: 1D k-stencil

We now illustrate this scheme using a 1D k-stencil kernel. We then apply the same principles to the finite field multiplication in Section 7.

1D k-Stencil. Given an input array $a_0, \ldots, a_{n-1}$, the output of a k-stencil kernel is an array $b_0, \ldots, b_{n-1}$ such that $b_i = \frac{\sum_{j=i-k}^{i+k} a_i}{2k+1}$, assuming $a_i = 0$ for $i < 0$ or $i \geq n$. k is also called a *window size*. Note that each input element is read $2k+1$ times during computation. Thus, any implementation must cache the input in order to exploit data reuse.

For simplicity we use $k = 1$, $\mathcal{W}$ =32 threads per warp.

Shared memory implementation. We consider the following implementation: (1) copy input from global memory into a temporary array in shared memory using all threadblock threads; (2) wait until the input is stored in shared memory; (3) compute one output element; (4) store the results in global memory.

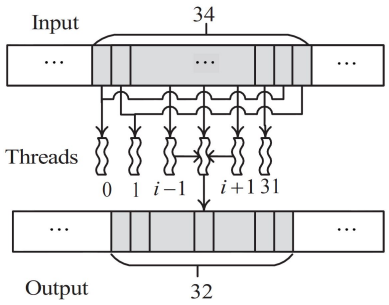

Figure 1: Input distribution in 1-stencil computation

Tid	0	1	2-29	30	31
Iteration 1			R(0,i) P(0)		
Iteration 2	R(0,1) P(1)	P(0,2) P(0)	R(0,$i+1$) P(0)	R(0,31) P(0)	R(1,0) P(0)
Iteration 3	R(0,2) P(1)	R(0,3) P(1)	R(0,$i+2$) P(0)	R(1,0) P(0)	R(1,1) P(0)

Table 1: Read (R) and Publish (P) operations in each iteration of the 1D 1-stencil computation. Tid denotes the thread index in a warp.

We follow the register cache methodology suggested above to eliminate shared memory accesses.

Step one: Identify warp inputs. Given that i is the index of the output element computed by thread 0 in a warp, the warp calculates the output elements $i, \ldots, i+31$, and depends on 34 input elements $i-1, \ldots, i+32$, denoted as `input` array.

Step two: Determine input distribution. We use a round-robin distribution of `input` arrays among the threads, as illustrated in Figure 1. In this scheme, `input[i]` is assigned to thread j=$i \mod 32$, where j is the thread index in the warp. Thread 0 and thread 1 each store two elements, while all the other threads store only one. We denote the first cached element as $r[0]$ and the second as $r[1]$. Observe that this distribution scheme mimics the data distribution across banks of shared memory.

Step three: Communication and computation. We identify three communication phases – one for each input element read by each thread. Table 1 lists all `Read` (R) and `Publish` (P) operations performed by each thread. `Read`(i, j) indicates a read from thread j of its element $r[i]$. The first communication phase is local, and provided for clarity.

We now merge Publish-Read tuples into `shuffle`(). At this point computations in a warp do not use shared memory. All that remains is to efficiently compute thread and register indexes in the `shuffle`() calls while avoiding divergence.

The complete implementation is in Listing 1.

3.2 Analysis

Bank conflicts and `shuffle`() conflicts. One of the main challenges of the register cache design is to transform the `Publish` and `Read` operations into `shuffle`() calls. In particular, if there are two or more threads performing `Read`(var, tid), such that tid is the same and var is different, this is called a *conflict*, since thread tid may fulfill these requests only in multiple `Publish` calls.

```
1 #define REGISTER_ARRAY_SIZE 2
2 #define FILTER_SIZE 1
3 __global__ void kstencilShuffle(
4     int* in,
5     int* out,
6     int size){
7   int threadInput[REGISTER_ARRAY_SIZE];
8   int threadOutput = 0, reg_idx, tid_idx;
9   int lindex = threadIdx.x & (WARP_SIZE - 1);
10  int gindex =
11    threadIdx.x + blockIdx.x * blockSize.x;
12 // PREFETCH. note: in is padded by FILTER_SIZE
13  int lowIdx = gindex - FILTER_SIZE;
14  int highIdx = lowIdx + WARP_SIZE;
15  threadInput[0] = input[lowIdx];
16  threadInput[1] = input[highIdx];
17
18 //First iteration - data available locally
19  threadOutput+=threadInput[0];
20
21 //COMMUNICATE + COMPUTE
22  reg_idx=(lindex==0)? 1 : 0 ;
23  tid_idx=(lindex+1) & (WARP_SIZE -1);
24  threadOutput+=
25    __shfl(threadInput[reg_idx],tid_idx);
26
27 //COMMUNICATE + COMPUTE
28  reg_idx =
29    (lindex == 0 || lindex == 1) ? 1 : 0 ;
30  tid_idx = (lindex+2) & (WARP_SIZE -1);
31  threadOutput+=
32    __shfl(threadInput[reg_idx],tid_idx);
33  output[gindex] = threadOutput / FILTER_SIZE;
34 }
```

Listing 1: 1-stencil implementation using the register cache.

One may argue that register cache conflicts are more likely than the bank conflicts in the original implementation. We argue that this is not the case. Consider the round-robin input distribution we used in the k-stencil example. This distribution mimics the distribution of data across the banks in shared memory, because, to the best of our knowledge, the number of banks in NVIDIA GPUs is the same as the number of threads in a warp. Thus, when using the round-robin distribution, the number of register cache conflicts will be exactly the same as the number of shared memory conflicts.

Moreover, register cache might make it possible to reduce the number of conflicts by using an alternative, application-optimized distribution of inputs. We leave this optimization question for future work.

Performance improvement over shared memory. The main benefits of register cache come from lower latency of `shuffle`() operations versus shared memory accesses [20], and higher bandwidth to registers compared to shared memory [27].

As an illustration, we compare the performance of shared memory and register cache implementations of the k-stencil kernel. We find that the register cache implementation achieves 64% higher throughput compared to the shared memory version for input sizes of 2^{27} elements.

Thread coarsening. One common technique in program optimizations is *thread coarsening* [3]. This technique increases the number of outputs produced by each thread, and thus enables some of the data to be reused across iterations by storing it in registers.

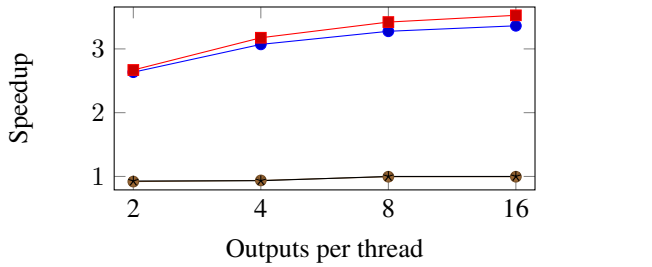

Figure 2: Speedup obtained from coarsening in the computation of $1 - Stencil$ and $7 - Stencil$ for register cache and shared memory implementation

In the case of the register cache, thread coarsening is sometimes *required* in order to achieve the desired performance improvements. The reason lies in the small number of threads sharing the cache. Since the register cache is limited to the threads of a single warp, only the inputs necessary for the warp threads are prefetched and cached. However, the input reuse might occur *across the warps*. For example, for the $k = 1$-stencil kernel, the value `array`[0] in warp i is the same as `array`[31] in warp $i-1$; however, both warps read it from the global memory. Thus, assuming the maximum of 32 warps in a threadblock, one threadblock in a register cache implementation performs $34 \times 32 = 1088$ global memory accesses, which is 6% more than the global memory accesses in a shared memory implementation with the same threadblock size. Moreover, the number of redundant memory accesses grows with k, reaching 88% for $k = 16$.

Thread coarsening helps reduce the effect of redundant global memory accesses. In Figure 2 we show the performance improvement due to computing more outputs per thread (2,4,8 and 16) for the implementations using register cache and shared memory, for different values of k. We see that the improvement due to thread coarsening is almost negligible for the shared memory version, but it is significant for the register cache. We note that with a single output per thread the shared memory version is actually 1.8-2 times faster than the one using register cache for all k (not shown in the graph). However with two and more outputs per thread, the register cache version is faster.

High data reuse. As with any cache, the effect of the register cache is amplified with higher data reuse. Figure 3 shows the relative performance of the register cache implementation of k-stencil over the shared memory implementation for different k, as a proxy for evaluating different amounts of data reuse. The speedup achieved by the register cache is about 10% higher for $k = 15$ than for $k = 1$. Each thread computes 16 outputs.

3.3 Limitations

Access pattern known at compile time. The register cache design may work only for a shared memory access pattern known at compile time. The main reason is that a thread must `Publish` its data exactly when the other threads need it, which requires static provisioning of the respective `shuffle`() calls. For memory accesses determined at runtime, such provisioning is impossible.

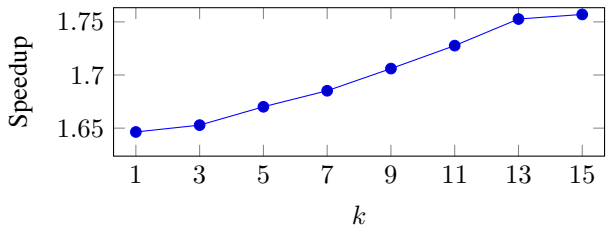

Figure 3: Speedup of the shuffle-based k-Stencil implementation over the shared memory-based implementation as a function of k

Register pressure. The register cache uses additional registers, and increases register pressure in a kernel. Even though recent NVIDIA GPUs increase the number of hardware registers per threadblock, the register pressure poses a hard limit on the number of registers available for caching and must be considered to avoid spillage.

4. FINITE FIELD MULTIPLICATION - ALGEBRAIC BACKGROUND

This section briefly reviews the basic elements of polynomial rings and Galois fields required by our implementation. For a thorough introduction to Galois fields, see, e.g., [22].

The ring of polynomials. $\mathrm{GF}(2)$ is a field with two elements $(0, 1)$, with addition ($\oplus$) and multiplication ($\odot$) performed modulo 2. A *polynomial over GF* (2) is an expression of the form $A(x) := \sum_{i=0}^{d} a_i x^i$, where $a_i \in \mathrm{GF}(2)$ and x is a formal variable; henceforth we simply call $A(x)$ a *polynomial* because all polynomials mentioned in this paper are over $\mathrm{GF}(2)$. The *degree* of A, denoted $\deg(A)$, is the largest index i such that $a_i \neq 0$. Addition and multiplication of polynomials (also called *ring addition and multiplication*) are defined in the natural way, i.e., for $B(x) = \sum_{i=0}^{m} b_i x^i$ with $m \geq d$ we have $A(x) \oplus B(x) = \sum_{i=0}^{m} (a_i \oplus b_i) x^i$ and $A(x) \odot B(x) = \sum_{j=0}^{d+m} x^j \cdot \bigoplus_{i=0}^{j} a_i \odot b_{j-i}$. The set of polynomials with the operations of addition and multiplication defined above, forms the *ring of polynomials over GF* (2), denoted $\mathrm{GF}(2)[x]$. Later, we reduce the problem of efficient multiplication in the *field* $\mathrm{GF}(2^n)$ to the problem of multiplying polynomials in the *ring* $\mathrm{GF}(2)[x]$.

The standard representation of a binary field. The most common way to represent $\mathrm{GF}(2^n)$, also used here, is via a *standard basis*, as described next. A polynomial $r(x) \in \mathrm{GF}(2)[x]$ of degree n is called *irreducible* if there is no pair of polynomials $g(x), f(x) \in \mathrm{GF}(2)[x]$ such that $r(x) = g(x) \odot f(x)$ and $\deg(g), \deg(f) < n$. Many irreducible polynomials exist for every degree n. (Later, a special class of irreducible polynomials will be used to speed up multiplication.) Having fixed an irreducible $r(x)$, for every pair A, B of polynomials of degree $< n$, there exists a unique polynomial C of degree $< n$ such that $r(x)$ divides $A(x) \odot$

$B(x) \oplus C(x)$ in the ring GF (2) $[x]$; i.e., there exists $C'(x)$ such that $A(x) \odot B(x) \oplus C(x) = r(x) \odot C'(x)$. Denote the transformation that maps the pair of polynomials $(A(x), B(x))$ to the polynomial $C(x)$ by $\otimes_r$, where r is used to emphasize that this transformation depends on the irreducible polynomial $r(x)$. The set of polynomials of degree $< n$, along with ring addition $\oplus$ and multiplication $\otimes_r$ defined above, is a *standard basis* representation[1] of GF (2^n). When the irreducible polynomial h is clear from context, we drop it and denote GF (2^n) multiplication simply by $\otimes$.

Example of multiplication in standard representation. In this example we show the field multiplication of two elements in GF $\left(2^4\right)$, using the standard representation induced by the irreducible degree-4 polynomial $r(x) := x^4 + x + 1$. Consider the two elements $A(x) = x + x^3$ and $B(x) = 1 + x^2$, represented in the standard basis by $a := (1010), b := (0101)$. To compute the 4-bit string $c = a \otimes_r b$ we work as follows:

- Compute the product $C'(x)$ of the two polynomials $A(x), B(x)$ in the ring GF (2) $[x]$, namely, $C'(x) := A(x) \odot B(x) = (x+x^3) \odot (1+x^2) = x + \cancel{2x^3} + x^5$ (middle term canceled because we work modulo 2).
- Compute the remainder $C(x)$ of the division of $C'(x)$ by $r(x)$; in our example $C(x) = x^2$ and one can verify that $\deg(C) < 4$ and $r(x) \odot x = C'(x) \oplus C(x)$, as defined above.

Thus, $a \otimes_r b = c$ where $c := (0100)$.

Field multiplication reduces to ring multiplication. The previous definitions and example show two main points that we exploit next. First, when multiplying two elements in the standard representation induced by $r(x)$, it suffices to (i) multiply polynomials in the ring GF (2) $[x]$ and then (ii) compute the remainder modulo $r(x)$. Second, the structure of $r(x)$ may influence the complexity of computing field multiplication.

5. SEQUENTIAL FINITE FIELD MULTIPLICATION

We now provide an efficient algorithm (Algorithm 1) for finite field multiplication, one that reduces field multiplication to a small number of polynomial multiplications; it requires a special standard basis, induced by a 2-*gapped polynomial*, defined next. In this section we use the following notation: given a polynomial $h(x) = \sum_{i=0}^{m} h_i x^i$, we define $h_a^b(x) = \sum_{i=a}^{b} h_i x^{i-a}$.

DEFINITION 1 (2-GAPPED POLYNOMIAL). *A polynomial $r(x)$ is* 2-Gapped *if the degree of its second-largest term is at most $\lfloor \frac{\deg(r(x))}{2} \rfloor$, i.e., if $r(x) = x^n + r_1(x)$ with $\deg(r_1(x)) \leq \lfloor \frac{n}{2} \rfloor$.*

Algorithm 1 performs GF(2^n) multiplication by reducing it to 3 GF (2) $[x]$ ring multiplications. Thus, *the performance of field multiplication is determined almost entirely by the complexity of multiplication of polynomials in the ring of polynomials*. Therefore, in the rest of the paper we focus on the problem of fast polynomial multiplication on GPUs.

[1]The term "basis" refers to the algebraic fact that the n elements $1, x, x^2, \ldots, x^{n-1}$ are linearly independent over GF (2), i.e., they form a basis for GF (2^n) over GF (2); cf. [22] for more information.

Algorithm 1 Multiplication in GF(2^n)

Input:
- $a(x), b(x)$ of degree at most $n-1$ in $\mathbb{F}_2[X]$.
- $r(x) = x^n + r_1(x)$, 2-gapped polynomial in $\mathbb{F}_2[X]$ of degree n.

Output: $h(x) = (a(x) \odot b(x)) \mod r(x)$

1: $h(x) \leftarrow a(x) \odot b(x)$
2: $h(x) \leftarrow h_0^{3n/2-1}(x) \oplus h_{3n/2}^{2n-1}(x) \odot r_1(x) \odot x^{n/2}$
3: $h(x) \leftarrow h_0^{n-1}(x) \oplus h_n^{3n/2}(x) \odot r_1(x)$
4: **return** $h(x)$

5.1 The CPU CLMUL instruction

Finite field arithmetic, in particular GF(2^n) multiplication, has received considerable attention (cf. [14, 9]) and has efficient CPU implementations in popular software libraries like NTL [26] and MPFQ [2]. Moreover, in large part because of the importance of GF (2^n) multiplication, Intel introduced in 2010 a dedicated CPU instruction set extension `CLMUL`, which performs GF (2) $[x]$ ring multiplication of polynomials of degree up to 64 in 7–14 cycles [10]. Both NTL and MPFQ use this dedicated instruction. This instruction can be used to multiply polynomials of higher degree, thereby supporting GF (2^n) multiplication for values $n > 64$ (cf. [5] for one such implementation).

5.2 Sequential polynomial multiplication

The complexity of polynomial multiplication has been extensively studied. The number of bit operations performed by naïve Algorithm 2 is $O\left(n^2\right)$. More sophisticated algorithms by Karatsuba [18] and by Schonhage and Strassen [25, 7] are asymptotically faster, requiring $O\left(n^{\log_2 3}\right)$ and $O(n \log n \log\log n)$ bit operations, respectively.

In this work we use the naïve Algorithm 2 because it is the fastest for polynomials of degrees below 1000 [19] and its simplicity makes it a prime starting point for study.

The following simple equation, which explicitly computes coefficients of the output polynomial, will be used later to balance work in the GPU.

$$c_k = \begin{cases} \sum_{i=0}^{k} a_i \cdot b_{k-i} & k \leq n-1 \\ \sum_{i=k}^{2n-2} a_{n-1+k-i} \cdot b_{i-n+1} & k > n-1 \end{cases} \quad (1)$$

6. PARALLEL POLYNOMIAL MULTIPLICATION

We consider the problem of performing multiplication of a large number of pairs of polynomials.

A naïve, purely data-parallel approach is to assign a single multiplication of two polynomials to one thread. Here, each polynomial of degree $n-1$ is represented as a bit array of

[2]http://mpfq.gforge.inria.fr/doc/doc.html

Algorithm 2 Naïve polynomial multiplication

Input:
$a(x), b(x)$ of degree at most $n-1$.
Output: $c(x) = a(x) \odot b(x)$
1: **for** $i = 0, \ldots, n-1$ **do**
2: $c_i \leftarrow 0$
3: **for** $j = 0, \ldots, i$ **do**
4: $c_i \leftarrow c_i \oplus a_j \odot b_{i-j}$
5: **for** $i = n, \ldots, 2n-2$ **do**
6: $c_i \leftarrow 0$
7: **for** $j = i, \ldots, 2n-2$ **do**
8: $c_i \leftarrow a_{n-1+i-j} \odot b_{j-n+1}$
9: **return** $c(x) = \sum_{i=0}^{2n-2} c_i \cdot x^i$

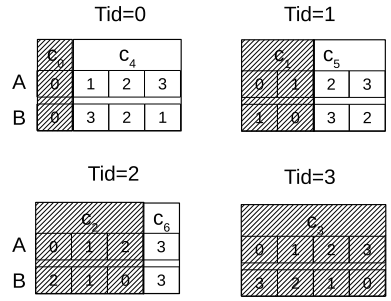

Figure 4: Illustration of the access pattern of the multiplication algorithm for GF(2^4) with $\mathcal{W}$ =4. Each frame encloses the indexes of rows in A and B accessed for computing the respective rows c_i specified on the top. Tid denotes the thread index in the warp.

size n, where the i^{th} element represents the coefficient of x^i in the polynomial.

This solution is highly inefficient, however. On a platform with $\mathcal{B}$-bit registers and ALUs, performing single-bit operations uses only $1/\mathcal{B}$ of the computing capacity. We therefore develop an alternative algorithm which eliminates this inefficiency.

6.1 Bit slicing

We reorganize the computation such that one thread performs bit-wise operations on $\mathcal{B}$ bits in regular registers, effectively batching multiple single-bit operations together. This technique, which packs multiple bits for parallel execution, is often called *bit-slicing* [29].

To employ bit-slicing for polynomial multiplication, we first introduce a new data structure, a *chunk*, to represent multiple polynomials, and then reformulate the multiplication algorithm using chunks.

DEFINITION 2 (CHUNK). *A chunk is an $n \times \mathcal{B}$ matrix M of bits that represents a set of $\mathcal{B}$ polynomials $P(i)$, $i \in \{0, \ldots, \mathcal{B}-1\}$ of degree less than n. We denote the j^{th} column in M by M^j, and the i^{th} row by M_i.M^j represents the coefficients of the j^{th} polynomial in the set. In other words, $A(i) = \sum_{i=0}^{n-1} M_i^j x^i$.*

To explain how to compute using chunks, we first consider polynomial addition. It is easy to see that it can be performed by bit-wise XOR of the respective rows of the input chunks A and B. Thus, a single $A_i \oplus B_i$ computes the i^{th} coefficients for all $\mathcal{B}$ output polynomials at once. Figure 5 shows two input chunks A, B, and the chunk representing their sum $A \oplus B$. Each chunk represents 4 polynomials of degree 3. For example, A^1 represents polynomial x^2. Figure 5 also shows an example of polynomial addition using chunks, assuming $\mathcal{B} = 4$.

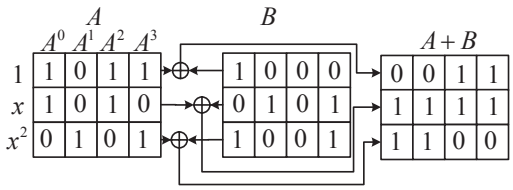

Figure 5: Polynomial addition in 4-bit chunks. Computing the output chunk requires 3 bit-wise XORs, each performing 4 concurrent $\oplus$ operations.

Similarly, it is straightforward to extend the single-bit polynomial multiplication Algorithm 2 to use chunks. This is done by replacing the references to individual bits in lines 2,4,6 and 8 with the references to chunk rows, and replacing single-bit operations with bit-wise operations.

6.2 Parallel polynomial multiplication using chunks

We show how to parallelize chunk-based polynomial multiplication. We seek a parallel algorithm that enables efficient, divergence-free execution by the threads of a single warp, which is the key to high performance on GPUs.

A simple parallelization whereby one thread computes one row in the output chunk is inefficient due to divergence among the threads. As we see from Eq. 1, different coefficients in the output polynomial require different numbers of computations. For example, computing the coefficient of x^2 requires only three $\oplus$ operations, while computing the one for x^3 requires four. Thus, different threads in a warp would perform different numbers of operations, resulting in divergence.

The key to achieve load-balanced execution among the threads is to realize that *pairs* of coefficients require exactly the same number of computations *in total*, as we show below.

Denote by $Add(k)$ and $Mul(k)$ the number of $\oplus$ and $\odot$ operations respectively to compute the k^{th} coefficient in the output polynomial. From Eq. 1 we derive that $Add(k) = \min\{k, 2n-2-k\}$, $Mul(k) = \min\{k+1, 2n-2-k+1\} = Add(k)+1$. Therefore $Add(k)$ and $Mul(k)$ are symmetric around $n-1$. Consequently, for each $0 \leq k < n$ $Add(k) + Add(k+n) = n-2$, $Mul(k) + Mul(k+n) = n$

We conclude that the number of computations needed to compute both coefficients k and $k+n$ together is exactly the same for all k. Therefore, allocating such pairs of coefficients to be computed by each thread will balance the load perfectly among the threads. Note that computations always interleave bitwise $\oplus$ and $\odot$ operations; therefore there is no divergence as long as the number of such operations in all

```
1  __global__ void multiply_shmem(
2        int* A, B, C,
3        int N)
4  {
5     __shared__ int sA[32];
6     __shared__ int sB[32];
7     int output=0;
8     int lindex = threadIdx.x & (WARP_SIZE - 1);
9
10 // PREFETCH
11    sA[lindex]=A[lindex];
12    sB[lindex]=B[lindex];
13    __syncthreads();
14
15    for (int i=0;i<=lindex;i++){
16       int a = sA[i];
17       int b = sB[lindex-i];
18       output ^= a&b;
19    }
20    C[lindex]=output;
21    output=0;
22    for (int i=lindex+1;i<N;i++){
23       int a = sA[i];
24       int b = sB[N-1+lindex-i];
25       output ^= a&b;
26    }
27    C[lindex+N]=output;
28 }
```

Listing 2: Multiplication of polynomials of degree 32 in a warp using shared memory.

```
1  __global__ void multiply_reg_cache(
2     int* A, B, C,
3     int N)
4  {
5    int a_cached, b_cached, output=0;
6    int lindex = threadIdx.x & (WARP_SIZE - 1);
7
8  // PREFETCH
9    a_cached=A[lindex];
10   b_cached=B[lindex];
11
12   for (int i = 0 ; i < N ; i++)
13   {    //COMMUNICATE
14     int a = __shfl(cached_a,i);
15     int b = __shfl(cached_b,lindex-i);
16   //COMPUTE
17     if (i <= lindex) output ^= a&b;
18   }
19   C[lindex]=output;
20   output=0;
21   for (int i = 0; i < N ; i++){
22     int a = __shfl(cached_a,i);
23     int b = __shfl(cached_b,N-1+lindex-i);
24
25     if (i > lindex) output ^= a&b;
26   }
27   C[lindex+N]=output;
28 }
```

Listing 3: Multiplication of polynomials of degree 32 in a warp using the register cache.

threads is the same.

In summary, our parallel polynomial multiplication algorithm allocates each thread in a warp to compute one or more pairs of rows $(k, k+N)$ in the output chunk. Each thread computes the coefficients of $\mathcal{B}$ polynomials at once, thanks to bit-slicing.

We illustrate the execution of the algorithm for GF$\left(2^4\right)$ and $\mathcal{W}=4$ threads per warp as an example in Figure 4.

Implementation. The implementation closely follows the algorithm. We dedicate one warp to compute $2\mathcal{W}$ rows in the output chunk C. All the rows in the input are accessed by all the threads, and therefore they are prefetched into shared memory. Figure 2 lists the implementation for a single warp, assuming $\mathcal{W}$ =N=32. For clarity we split the implementation into two separate loops (line 15 and 22), each computing one output row. This leads to divergence in practice, so in the real implementation these two loops are merged.

Limitations. The algorithm achieves divergence-free execution for polynomial multiplication in GF$\left(2^N\right)$ when $N|\mathcal{W}$, i.e., 32, 64, 96. We leave the question of efficient multiplication of polynomials of other degrees to future work.

7. POLYNOMIAL MULTIPLICATION VIA REGISTER CACHE

In this section we apply the register cache methodology presented in Section 3 to speed up ring multiplication (Listing 3) and compare it (here and later) to the less efficient and simpler shared memory implementation (Listing 2). To describe the register cache optimizations, we focus on a single warp performing multiplication of polynomials of degree n=$\mathcal{W}$ =32. We then discuss the application of this method to polynomials of higher degree.

We start with the shared memory implementation described in Section 6.2.

Step one: Identify warp inputs in shared memory. Since each warp is dedicated to the calculation of a single product of two chunks, each warp reads only its input chunks.

Step two: Distribute inputs among warp threads. The rows in chunks are distributed in a round-robin fashion across the warp threads. For each of the two input chunks, thread ℓ stores all the chunk rows t such that $\ell \equiv t \mod w$. Conveniently, since $\mathcal{W}=n$, thread i stores rows A_i and B_i of the respective chunks.

Step three: Split the algorithm into communication and computation steps. Each thread communicates with the other threads to obtain the operands of each $\odot$ operation. Therefore, each $\odot$ is a computation step that is preceded by a communication step in which the operands are received. We refer to two such steps together as *an iteration*, because they correspond to one iteration of the loops in lines 15 and 22 in Listing 2.

We first determine the data accessed by each thread. We derive this from the accesses to shared memory in lines 16-17 and 23-24 in Listing 2. Due to the round-robin data distribution we use, and since the number of rows in each chunk equals the number of threads, the indexes in shared memory coincide with the warp indexes of the threads holding the data.

Now we derive which data must be published by each thread in each iteration. Figure 4 is useful to reason about this. We see that the value of A_i, stored in thread i, is needed by all the threads only in iteration i, and hence each thread must publish it in iteration i. B_i, however, is read by differ-

ent threads in different iterations. For example, B_0 is used by thread 0 in the first iteration, thread 1 in the second, and so on. Thus, thread i must publish B_i in each iteration.

The computation in each iteration remains the same as in the shared memory version.

Replacing each communication step with shuffles. To use `shuffle()`, we must align `Read` and `Publish` operations in each communication step. To simplify, we consider the case in which we first align all accesses to B and then to A.

Aligning accesses to B is straightforward, because (1) each thread publishes its single cached value and reads one value in every iteration, and (2) no two threads require two different values at once from the same thread (which would result in a conflict).

The accesses to A cause a problem, because each thread publishes only in one iteration, but reads in each iteration. The solution is to simply duplicate the `Publish` operation to each iteration, even though it is redundant.

The complete algorithm is presented in Listing 3.

8. EXTENDING TO POLYNOMIALS OF LARGER DEGREES

We now extend the register cache-based multiplication implementation described in the previous section to polynomials of larger degrees. Doing so requires us to cope with the challenge of limited register space.

The shared memory algorithm in Listing 2 can be extended to up to $n = 1024$ by adding more warps, each using the same code structure. The register cache, however, is applicable only within a single warp. Therefore such a simple extension does not work for the optimized algorithm.

However, extending the register cache for higher degree polynomials is problematic in other ways as well. Caching these large polynomials requires more register space. Thus, at a certain threshold n_0, high register pressure results in register spillage to global memory, thereby rendering the register cache method described above inapplicable. We found empirically that the threshold is $n_0 = 64$.

In order to efficiently multiply polynomials of degree $n > 64$, we develop a hybrid solution that uses the efficient register cache-based implementation for multiplying polynomials of lower degree. The idea is to use the lower-degree multiplication as a building block for multiplying polynomials of higher degrees, at the expense of employing shared memory.

The full description of this algorithm is omitted for lack of space. But we now explain the main idea behind it, by showing how to multiply degree-64 polynomials using multiplication of degree-32 polynomials as a building block.

Let $a(x) = \sum a_i x^i$ and $b(x) = \sum b_i x^i$ be two polynomials of degree 64 that we wish to multiply. Denote the efficient procedure for multiplying two polynomials of degree 32 by `mult32()`. We can represent $a(x) = a_0(x) + x^{32}a_1(x)$, where $a_0(x) = \sum_{i=0}^{31} A_i x^i$ and $a_1 = \sum_{i=32}^{63} x^i$. Observe that a_0 and a_1 are two polynomials of degree at most 31. Using the same representation for $b(x)$, we obtain $a(x) \odot b(x) = (a_0(x) + x^{32}a_1(x)) \odot (b_0(x) + x^{32}b_1(x)) = \texttt{mult32}(a_0(x), b_0(x)) + x^{32}\texttt{mult32}(a_1(x), b_0(x)) + x^{32}\texttt{mult32}(a_0(x), b_1(x)) + x^{64}\texttt{mult32}(a_1(x), b_1(x))$.

Version	Throughput (mult/s $\times 10^9$)	Shared memory accesses	Reg/Thread
shmem	1.04	16384	25
mult32	2.7	512	30
rcache	3.6	0	32

Table 2: Performance of three different implementations of 64-degree polynomial multiplication.

There are many possible implementations of this idea and those we are aware of use shared memory. We choose to implement one such solution that uses two warps. The first warp computes `mult32` (a_0, b_0) and `mult32` (a_1, b_0), and the second one computes `mult32` (a_0, b_1) and `mult32`(a_1, b_1). Since the input is reused across the warps, it is stored in shared memory. In addition, each warp stores its output in shared memory, so the two warps can combine the results of `mult32` (a_0, b_1) and `mult32` (a_1, b_0).

We use the same principle to implement multiplication for polynomials of higher degree.

8.1 Comparison of the different designs

We would like to compare the relative speedup offered by the hybrid algorithm over the purely shared memory implementation, and over the implementation that uses the register cache only. Comparing these three designs is possible only for $n \leq 64$ because, as mentioned, register pressure in the register cache version results in register spillage.

In our implementation, the naive shared memory version runs in two warps. The hybrid `mult32`-based implementation uses the `mult32` function internally, and uses shared memory to share input and intermediate outputs between warps. Finally, the optimized degree-64 multiplication uses register cache natively, without shared memory. In this implementation each thread stores 4 input coefficients and produces 4 outputs.

The results of the comparison are presented in Table 2 and demonstrate the benefits of using register cache. We observe that the shared memory (shmem) implementation is about 3.5 times slower than the one using register cache (rcache). The hybrid version (mult32) achieves 2.6 times faster execution over shmem, and about 30% slower than the optimal rcache version.

These results also indicate that the best building block for the hybrid algorithm is the multiplication kernel of the largest degree that fits in the register cache. Therefore, we use n-64 polynomial multiplication and evaluate its performance in Section 9.

8.2 Application to larger fields

The shared memory based multiplication requires $16n$ bytes of shared memory. In a GPU with up to 48KB of shared memory per threadblock for full occupancy (as NVIDIA Titan-X), we are limited to fields of size $< 2^{3072}$. With the register cache we use half the amount of shared memory, and therefore can implement multiplication in fields as large as $\text{GF}(2^{6144})$.

However, we do not implement it for fields larger than

GF (2^{2048}). For larger fields the hybrid algorithm outlined here with asymptotic running time $O(n^2)$ becomes relatively inefficient when compared to the more sophisticated Karatsuba algorithm, as detailed in Section 9.

9. EVALUATION

We evaluate the complete implementation of GF(2^n) multiplication. The source code is available online [3]. We incorporate the algorithms in Section 7 into the finite field multiplication implementation according to Algorithm 1.

Methodology. We use GeForce® GTX TITAN-X GPU, and a Supermicro Server with 2x6 Intel® Xeon® E5-2620 v2 @ 2.10GHz CPUs with 64GB of RAM. For each measurement we perform five executions, remove the highest and lowest results, and compute the average of the remaining three. We observe negligible standard deviation, less than $< 4\%$. Hyperthreading and CPU power management are disabled to achieve reproducible CPU performance. Each experiment uses random data for its input. As a CPU baseline we use NTL version 8.1.2 [26], which is a highly-optimized single-core CPU library for finite field arithmetics that uses `CLMUL` CPU intrinsics for polynomial multiplication.

Speedup over CPU for GF(2^{32}) and GF(2^{64}). Our implementation for GF(2^{32}) and GF(2^{64}) employs optimized register cache implementations of n=32 and n=64 polynomial multiplication respectively. We emphasize that we apply the same optimizations the NTL does when 2-gapped polynomials are used, and that the NTL implementation is based on the `CLMUL` instruction.

Figure 6 shows the results. The GPU implementations for GF(2^{64}) and GF(2^{32}) are up to **99** $\times$ and **138** $\times$ faster than NTL's CPU multiplication for inputs exceeding 2^{26} elements.

We observe that the speedups are not constant. The reason lies in the variability in the NTL performance, which drops by about 15% for larger inputs. The GPU implementation performance keeps rising until it plateaus out for inputs exceeding 2^{25} elements.

The peak throughputs of GPU implementations are **3.15** and **2.09** billion finite field multiplications per second for GF(2^{32}) and GF(2^{64}) respectively. Note that these throughputs are slightly lower than the throughput of the respective polynomial multiplication, because finite field multiplication involves multiple polynomial multiplications.

Register cache vs. shared memory. We compare two implementations for multiplication in GF(2^{64}): with shared memory and with register cache. This experiment seeks to evaluate the impact of our register cache optimization on the end-to-end application performance. We observe that the register cache version is 50% faster than the shared memory version. As expected, the performance boost is smaller than in the pure polynomial multiplication case reported in Table 2.

Performance for larger fields. We evaluate the performance of the finite field multiplication in fields of higher degrees.

[3] https://github.com/HamilM/GpuBinFieldMult

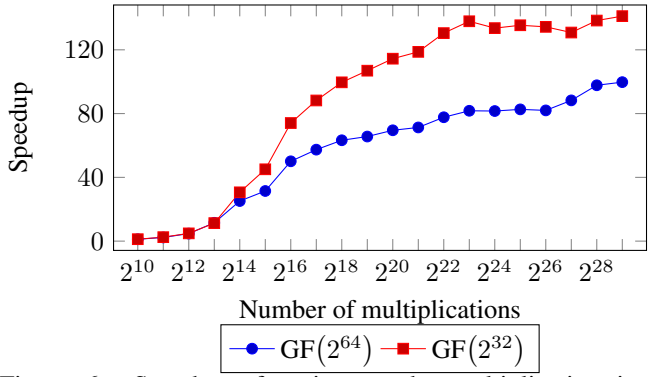

Figure 6: Speedup of register cache multiplication in GF(2^{64}) and GF(2^{32}) over NTL

Here we incorporate our hybrid implementation for polynomial multiplication described in Section 8, using the n=32 polynomial multiplication as its building block. We measure the performance for fields from GF(2^{32}) to GF(2^{2048}). We use 2^{23} elements per input.

Figure 7 shows the speedup of our implementation over NTL. We achieve significant speedups for smaller fields, but when fields grow larger our speedup diminishes (to 2.17$\times$ in GF(2^{2048})). The reasons are found in the NTL implementation. For fields smaller than GF(2^{64}), NTL uses the `CLMUL` intrisics, which allow only multiplication of n=64 degree polynomials; the implementation is therefore inefficient for these fields. Our GPU implementation does not suffer from this limitation. However, for larger fields NTL uses a different hybrid algorithm (Karatsuba), which is asymptotically faster than the quadratic algorithm we use. The problem of implementing the Karatsuba algorithm on GPUs is in the difficulty to balance the load across threads. We leave the implementation of a GPU Karatsuba for future work.

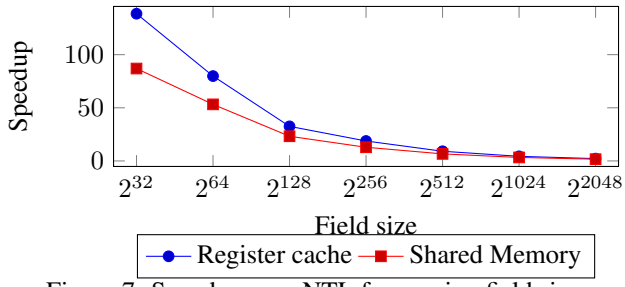

Figure 7: Speedup over NTL for varying field sizes

Performance for other fields. Figure 8 shows the performance of our GPU implementation for GF(2^N) where $N \neq 2^n$. As expected, we observe the step function, where in each step the inputs are processed by the same number of warps. The number of warps in our implementation employed in GF(2^N) is $\lceil \frac{N}{64} \rceil$.

Considering alternative CPU implementations. In all our experiments we use a single-threaded NTL implementation for CPU as the performance baseline. NTL natively supports multiplication of a single pair of elements and uses `CLMUL` instruction. One could argue, however, that extending NTL

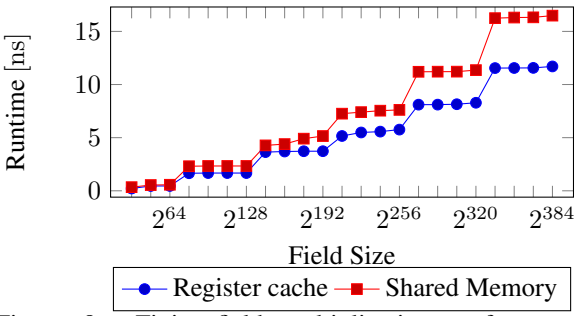

Figure 8: Finite field multiplication performance for GF(2^N) where N is not a power of 2.

to support multiplication of many pairs in a batch, as we do in GPUs, might open additional optimization opportunities, e.g., bit-slicing techniques like those proposed in Section 6. Thus, it would become possible to use the AVX vector instruction set instead of `CLMUL`, potentially improving NTL performance.

We now show why `CLMUL` implementation is superior. In the AVX instruction set [10] a single 512-bits wide AND and XOR takes 1 cycle each. Therefore, using our bit-slicing algorithm, we can multiply 512 pairs of polynomials of degree 64 in $2 \times (64^2) = 8192$ cycles. Note that this estimate is rather optimistic, as we ignore the time to reorganize the input bits to allow vectorized execution. On the other hand, each CLMUL instruction multiplies a single pair of polynomials of degree 64 in 3.5 cycles (latency 7 cycle, throughput=2) [10]. Therefore, 512 polynomials can be multiplied in $3.5 \times 512 = 1792$ cycles alone, much faster then the bit-sliced AVX-based implementation.

10. RELATED WORK

2-gapped polynomials. The CPU implementation of NTL [26] for the multiplication in binary fields uses the `CLMUL` [12] instruction and employs 2-gapped polynomials to replace reduction with multiplications. We apply a similar algorithm in our work.

SIMD and bit-slicing. The CPU SIMD instructions have been used to perform bit-slicing to parallelize GF(2^n) multiplication [16]. Their implementation, however, is limited to small fields (up to GF(2^{32})). The GPU architecture suits SIMD computation and can provide the same functionality as the CPU SIMD instruction set [24]. The proposed implementation is, however, also limited to small fields (e.g GF(2^{16})). Our implementation applies to larger fields.

Finite field multiplication on GPUs. The previous works [16, 24] are limited to fields of size smaller than 2^{32}. Particularly, Plank [16] shows a CPU implementation that deals with computing a product of multiple elements by a single scalar, using scalar-dependent pre-computed lookup tables. Our work focuses on multiplying many *pairs of arbitrary elements*, therefore the lookup table approach is inapplicable.

Cohen et al. [2] describes an implementation of finite field multiplication in specific binary fields. The performance reported in their paper is 3-orders of magnitude slower than the performance reported in our work, and their implementation would benefit from bit slicing, register cache and reduced synchronization techniques presented here.

An implementation of finite field multiplication on GPUs over $GF(q)$ for some specific large NIST primes q is discussed in [17]. Our implementation, however, is optimized for binary fields in a scalable fashion to achieve a generic implementation for a large variety of field sizes.

Register-based optimizations. The benefits of reusing data in registers on GPUs to boost performance are well known. Volkov and Demmel [27] present GPU implementations of LU decomposition and SGEMM.

Enfedaque et al. [21] show how to implement the DWT (discrete wavelet transform) of an image of varying sizes where each warp calculates a different part of the output. They also show that shuffle-based communication achieves better results when the data each warp fetches from global memory is reused more times, as also confirmed by our results (cf. Section 3).

Davidson and Owens [1] suggest a method called *register packing* to reduce shared memory traffic in GPU when dealing with a downsweep patterned computation, by performing some parts of the computation in registers.

Catanzaro et al. [4] show a shuffle-based implementation for SIMD architectures, including the GPU. They discuss the benefits of the instruction for reducing shared-memory bandwidth and show the relation to the Array of Structs – Struct of Arrays transforms.

nVIDIA's Kepler Tuning Guide [20] stresses the benefits of registers over shared memory in terms of latency and capacity. The `shuffle` instruction is suggested as an alternative for the use of shared memory in some cases.

We leverage the lessons learned in the previous work, and take one additional step by suggesting a register cache design methodology for reducing shared memory accesses to the input data. We demonstrate the application of this methodology on a challenging case of finite field multiplication in binary fields, and show that it achieves significant performance benefits.

Acknowledgements

Mark Silberstein is supported by the Israel Science Foundation (grant No. 1138/14), the Israeli Ministry of Science, and the Israeli Ministry of Economics.

11. REFERENCES

[1] A. Davidson, and J. D. Owens. Register packing for cyclic reduction: A case study. In *Proceedings of the Fourth Workshop on General Purpose Processing on Graphics Processing Units*, pages 4:1–4:6. ACM, 2011.

[2] A. E. Cohen and K. K. Parhi. GPU Accelerated Elliptic Curve Cryptography in $GF(2^m)$. In *IEEE 53rd International Midwest Symposium on Circuits and Systems*, pages 57–60, Aug 2010.

[3] A. Magni, C. Dubach, and M. F. P. O'Boyle. A Large-scale Cross-architecture Evaluation of

Thread-coarsening. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, pages 11:1–11:11. ACM, 2013.

[4] B. Catanzaro, A. Keller, and M. Garland. A decomposition for in-place matrix transposition. *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 193–206, 2014.

[5] C. Su, and H. Fan. Impact of Intel's new instruction sets on software implementation of GF(2)[x] multiplication. *Inf. Process. Lett.*, 112(12):497–502, June 2012.

[6] D. B. Kirk, and W. W. Hwu. *Programming Massively Parallel Pocessors: A Hands-on Approach*. Newnes, 2012.

[7] D. G. Cantor, and E. Kaltofen. On fast multiplication of polynomials over arbitrary algebras. *Acta Informatica*, 28(7):693–701, 1991.

[8] E. Ben-Sasson, and M. Sudan. Short PCPs with polylog query complexity. *SIAM Journal on Computing*, 38(2):551–607, 2008. Preliminary version appeared in STOC '05.

[9] E. D. Win, A. Bosselaers, S. Vandenberghe, P. D. Gersem, and J. Vandewalle. A fast software implementation for arithmetic operations in GF(2^n). In *Proceedings of the International Conference on the Theory and Applications of Cryptology and Information Security: Advances in Cryptology*, pages 65–76. Springer-Verlag, 1996.

[10] A. Fog. Lists of instruction latencies, throughputs and micro-operation breakdowns for Intel, AMD and VIA CPUs. Available at http://www.agner.org/optimize/instruction_tables.pdf, 1996-2016. [Online; accessed 28-Mar-2016].

[11] G. L. Steele Jr., and J. B. Tristan. Using butterfly-patterned partial sums to optimize GPU memory accesses for drawing from discrete distributions. *CoRR*, abs/1505.03851, 2015.

[12] G. Shay, and M. E. Kounavis. Intel(R) carry-less multiplication instruction and its usage for computing the GCM mode - rev 2.02. Intel Corporation, April 2014.

[13] J. Daemen, and V. Rijmen. AES proposal: Rijndael. Available at http: //jda.noekeon.org/JDA_VRI_Rijndael_V2_1999.pdf, 1998. [Online; accessed 28-Mar-2016].

[14] J. L. Massey, and J. K. Omura. Computational method and apparatus for finite field arithmetic. US patent number 4587627. May 1986.

[15] J. Luo, K. D. Bowers, A. Oprea, and L. Xu. Efficient software implementations of large finite fields GF(2^n) for secure storage applications. *Trans. Storage*, 8(1):2:1–2:27, February 2012.

[16] J. S. Plank, K. M. Greenan, and E. L. Miller. Screaming fast Galois field arithmetic using Intel SIMD instructions. In *11th USENIX Conference on File and Storage Technologies*, pages 298–306, February 2013.

[17] K. Leboeuf, R. Muscedere, and M. Ahmadi. High performance prime field multiplication for GPU. In *IEEE International Symposium on Circuits and Systems*, pages 93–96, May 2012.

[18] A. Karatsuba and Y. Ofman. Multiplication of Many-Digital Numbers by Automatic Computers. Doklady Akad. Nauk SSSR145, 293-294, 1962. Translation in Physics-Doklady 7, 595-596, 1963.

[19] M. Arabi. *Comparison of Traditional, Karatsuba and Fourier Big Integer Multiplication. B.Sc. Thesis*. University of Bath, May 2005.

[20] nVidia. Kepler Tuning Guide. http://docs.nvidia.com/cuda/kepler-tuning-guide/index.html, 2015. [Online; accessed 26-Jan-2016].

[21] P. Enfedaque, F. Aulí-Llinàs, and J. C. Moure. Implementation of the DWT in a GPU through a register-based strategy. *IEEE Transactions on Parallel and Distributed Systems*, 26(12):3394–3406, Dec 2015.

[22] R. Lidl and H. Niederreiter. *Finite Fields*. (2nd ed.), Cambridge University Press, 1997.

[23] S. Ashkiani, A. Davidson, U. Meyer, and J. D. Owens. GPU Multisplit. *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 12:1–12:13, 2016.

[24] S. Kalcher, and V. Lindenstruth. Accelerating Galois field arithmetic for Reed-Solomon erasure codes in storage applications. In *IEEE International Conference on Cluster Computing*, pages 290–298, Sept 2011.

[25] A. Schonhage and V. Strassen. Schnelle multiplikation grosser zahlen. *Computing*, 7(3-4):281–292, 1971.

[26] V. Shoup. NTL: A library for doing number theory. Avilable at http://www.shoup.net/ntl, 2003. [Online; accessed 28-Mar-2016].

[27] V. Volkov, and J. W. Demmel. Benchmarking GPUs to tune dense linear algebra. In *International Conference for High Performance Computing, Networking, Storage and Analysis, Nov 2008.*, pages 1–11.

[28] V. Volkov. Better performance at lower occupancy. *Proceedings of the GPU Technology Conference*, 2010.

[29] Wikipedia. Bit slicing — Wikipedia,. [Online; accessed 27-Mar-2016].