Usable assembly language for GPUs: a success story

Daniel J. Bernstein[*], Hsieh-Chung Chen[†], Chen-Mou Cheng[‡], Tanja Lange[§],
Ruben Niederhagen[¶§], Peter Schwabe[‖¶], Bo-Yin Yang[¶]

[*] Department of Computer Science
University of Illinois at Chicago
851 S. Morgan Street, Chicago, IL 60607-7053, USA
Email: djb@cr.yp.to

[†] Harvard University
Email: kc@crypto.tw

[‡] Department of Electrical Engineering
National Taiwan University
1, Section 4, Roosevelt Road, Taipei 10617, Taiwan
Email: doug@crypto.tw

[§] Department of Mathematics and Computer Science
Eindhoven University of Technology
Den Dolech 2, 5600 MB Eindhoven, Netherlands
Email: tanja@hyperelliptic.org

[¶] Institute of Information Science
Academia Sinica
No 128, Academia Road, Section 2, Nankang, Taipei 11529, Taiwan
Email: by@crypto.tw, ruben@polycephaly.org

[‖] Research Center for Information Technology Innovation
Academia Sinica
No 128, Academia Road, Section 2, Nankang, Taipei 11529, Taiwan
Email: peter@cryptojedi.org

***Abstract*—The NVIDIA compilers `nvcc` and `ptxas` leave the programmer with only very limited control over register allocation, register spills, instruction selection, and instruction scheduling. In theory a programmer can gain control by writing an entire kernel in van der Laan's `cudasm` assembly language, but this requires tedious, error-prone tracking of register assignments.**

This paper introduces a higher-level assembly language, `qhasm-cudasm`, that allows much faster programming while providing the same amount of control over the GPU. This language has been used successfully to build a 90000-machine-instruction kernel for a computation described in detail in the paper, the largest public cryptanalytic project in history. The best GTX 295 speed that has been obtained for this computation with `nvcc` and `ptxas` is 25 million iterations per second; the best GTX 295 speed that has been obtained with `qhasm-cudasm` is 63 million iterations per second.

I. Introduction

Decades of advances in the design of optimizing compilers have *reduced*, but have not *eliminated*, the need for some performance-critical functions to be written in assembly language. For high-performance implementations in C it is common practice to write the critical parts in assembly.

This paper introduces a new assembly language `qhasm-cudasm` for programming Graphics Processing Units (GPUs), specifically NVIDIA's Tesla-architecture GPUs. There is no analogue to assembly in CUDA, NVIDIA's standard GPU programming toolkit. NVIDIA's "PTX" is labelled as an assembly language, but it is in fact compiled and hides too many machine details to be usable for writing cutting-edge code. We started from van der Laan's `cudasm`, a true assembly language for Tesla GPUs; we completely redesigned the syntax for readability and incorporated a powerful register allocator from `qhasm` [3], ultimately obtaining a *usable* assembly language for GPUs.

We have used `qhasm-cudasm` successfully to produce highly optimized code for a major cryptanalytic computation, the "ECC2K-130" computation, an order of magnitude larger than the RSA-768 factorization announced in [11]. This paper describes this implementation in detail as a case study for `qhasm-cudasm`. Finishing the computation in two years with the fastest CPU software developed would require 1595 quad-core (3.2 GHz AMD Phenom II X4 955) PCs, but would require just 534 GTX 295 graphics cards with our software, or just 229 cost-optimized PCs each having a quad-core CPU and two GTX 295 graphics cards.

Our software was optimized for and tested on a GTX 295 graphics card containing two 1.242GHz G200b GPUs. We later tried the same software on the T10 GPUs (in Tesla S1070-500 units) in TeraGrid's Lincoln cluster, similar GPUs in the NCF/SARA cluster, and the FX 5800 GPUs in TeraGrid's Longhorn cluster; documentation indicates that these GPUs have essentially the same microarchitecture as the GTX 295, except for small differences in clock speed, and as expected we saw very similar cycle counts. Of course, assembly languages need to be redone for each new architecture, but massive speedups are ample justification for this effort.

A. *Cores and multiprocessors*

Each of the targeted GPUs contains 30 "multiprocessors"; each multiprocessor contains 8 "streaming processors" and a few auxiliary units; each "streaming processor" performs, typically, one 32-bit operation per cycle. **Warning:** NVIDIA documentation often refers to each "streaming processor" as a "core." We find that terminology misleading: GPU multiprocessors are much more closely analogous to CPU cores, and GPU streaming processors are much more closely analogous to CPU ALUs. **In this paper we refer to multiprocessors as cores and streaming processors as ALUs.**

This paper describes a computation running on one core of a GPU. Cycle counts in the paper are cycle counts for a single GTX 295 core. Of course, to make full use of the GPU we actually run 30 independent computations on the 30 cores of the GPU, i.e., 60 independent computations on the GTX 295.

B. *Threads*

There is an additional level of parallelism within each GPU core: the core runs many threads of computation simultaneously. The number of threads is chosen by the programmer, but is limited by various resources shared among threads.

There are many cycles of latency between successive operations in a thread, and standard advice is to run at least 192 threads per core to hide this latency. For several reasons we chose to run only 128 threads per core in the particular computation discussed in this paper. The penalty for using only 128 threads turns out to be quite small: experiments show that 128 threads performing arithmetic operations in registers can keep all of the ALUs busy in a GPU core, and that occasional loads and stores from "shared memory" can be handled by the "special function units" in the GPU core without slowing down the ALUs. One important reason to limit the number of threads, although not the only reason, is that all threads on a core have to share a total of 16384 registers; 128 threads are each given 128 registers (minus a few special registers), and we make good use of those registers. A smaller number of threads would linearly reduce the ALU utilization and would not provide more registers per thread: the hardware does not allow one thread to address more than 128 registers.

C. *Bitslicing*

The particular computation targeted in this paper consists of a series of iterations summarized in Section II. Each iteration is a straight-line sequence of approximately 70000 bit operations. The bit operations are two-input ANDs (the output is 1 if both inputs are 1, otherwise 0) and two-input XORs (the output is 1 if the inputs are different, otherwise 0).

We carry out 128 independent iterations in parallel on a single GPU core, with no communication between the iterations. We apply each bit operation as a 128-way SIMD operation. GPU instructions, like CPU instructions, are word-oriented, so we adopt a standard strategy called "bitslicing": the 32 consecutive bits in a 32-bit register, or in a 32-bit memory location, are from 32 separate iterations. A 128-way XOR is then built straightforwardly from four 32-bit XORs, and a 128-way AND is built straightforwardly from four 32-bit ANDs.

We emphasize that these are 128 *bit* operations. For comparison, a single 32-bit instruction followed by 128 threads in parallel is carrying out 4096 bit operations. In other words, 128-way parallelism is only 1/32nd of the parallelism required to keep 128 threads busy. We would like to carry out many more independent iterations in parallel, but we are limited by the amount of shared memory available on a GPU core, especially for multiplication; see Section IX.

II. Target: the ECC2K-130 challenge

Many cryptographic implementations base their security on the hardness of the Discrete Logarithm Problem (DLP) on elliptic curves. Cryptanalysis, the science of breaking cryptography, plays an important role in determining the hardness of the DLP in relation to the different ways of choosing these curves—critical information for users who want to choose curves that balance efficiency and security.

In general, the hardness of the DLP grows with the size of the curve and there are several types of curves that are particularly hard to break. To give good estimates for the cost of breaking the DLP, challenge problems are posed at various sizes and for the different types of curves. From the time necessary to break these challenges one can extrapolate the costs for breaking larger systems and estimate how much computational effort is necessary to break a real-world system; this then translates into how much money is necessary to buy the hardware necessary to break actual cryptographic implementations. Of course the cost estimates depend on the type of hardware—customized ASICs are likely to have a better price-performance ratio than off-the-shelf PCs but also require more specialized knowledge and come at a high initial cost for designing and fabricating the chip. Challenges put no restrictions on the hardware used but so far all breaks of challenges in public-key cryptography have used standard CPUs with ample RAM per thread, such as the Intel Core 2 or the PlayStation 3.

The computation described in this paper is part of a big cryptanalytic project, namely the breaking of the Certicom challenge ECC2K-130 [7]. The ECC2K-130 challenge was posed in 1997 and remains unbroken today. Our estimates are that about 2^{77} bit operations are necessary to break this challenge, which just barely brings this computation into reach for academic teams. Full details about the challenge and the mathematical background are described in [1]. That paper also gives an overview of how the core part of the ECC2K-130 computation can be implemented on various platforms, ranging from standard CPUs to FPGAs and ASICs. The paper contains a section on GPUs but achieves worse performance on an entire GTX 295 than on a PlayStation 3, even though the GTX 295 is theoretically capable of performing almost eight times as many bit operations per second (10× as many cores, 2× as many bit operations per cycle per core, 0.388× the clock speed). More details on the fast implementation that we use here to illustrate the features of `qhasm-cudasm` are given in [4].

The computation is embarrassingly parallel and does not require much communication or storage. The challenge for the implementor is to perform as many meaningful bit operations per cycle as possible. Unfortunately, as described in the next section, the available compiler tools were not able to schedule the instructions and memory accesses in a way suitable to keep the arithmetic units of the GPU busy.

Our ECC2K-130 computation works with two representations of the finite field $\mathbf{F}_{2^{131}}$: a polynomial-basis representation introduced in [5], and a standard normal-basis representation. Multiplications naturally begin with polynomial-basis inputs but can efficiently produce outputs in either polynomial basis or normal basis. Squarings are most efficient in normal-basis representation.

The computation consists of a large number of iterations, with an essentially unlimited number of iterations to perform in parallel. We repeat here the description of the iteration function from [5, Section 5]. The input to an iteration is a pair (x,y) of elements of $\mathbf{F}_{2^{131}}$ satisfying certain conditions, notably that x has even Hamming weight w in normal basis. The output is another pair (x',y') defined by

$$\begin{aligned} x' &= \lambda^2 + \lambda + x + x^{2^j}, \\ y' &= \lambda(x + x') + x' + y, \end{aligned}$$

where $j = 3 + ((w/2) \bmod 8)$ and $\lambda = (y + y^{2^j})/(x + x^{2^j})$.

Each iteration involves three multiplications to compute the reciprocal of $x + x^{2^j}$, two further multiplications, four conversions from normal basis to polynomial basis, and many squarings and additions. See [5] for further details and analysis of the number of bit operations per iteration. Major subroutines in our software include `ppn`/`ppp` for a multiplication producing normal-basis/polynomial-basis output respectively, `multprep` for a basis conversion, `hamming` for a Hamming-weight computation, `add` for an addition, and `sq` for a squaring. These operations are described in later sections when we explain details of their implementation in the new framework of our usable assembly language.

III. Trouble with the CUDA toolchain

The hardware architecture of GPUs is expected to change even faster than the architecture of CPUs, not only in respect to the number of cores and sizes of caches but also in respect to the instruction set and register file organization. However, NVIDIA has committed to keep CUDA programs forward-compatible, ensuring that code written today runs on new video cards of tomorrow. NVIDIA covers a wide range of GPUs over several generations by using layers of abstraction.

We briefly recap the normal compilation process supported by NVIDIA. The programmer writes software in a C-like language, either CUDA or OpenCL. The NVIDIA compiler `nvcc` compiles this software into instructions for a pseudo-machine called PTX ("parallel thread execution"). The second-stage NVIDIA compiler `ptxas` (normally invoked implicitly by `nvcc`) reads PTX code and produces actual GPU machine language in an undocumented format called `.cubin`. The CUDA driver loads the `.cubin` file onto a GPU and runs it. The `.cubin` format was text before CUDA 3.0, but was then replaced by a binary format requiring less parsing.

Writing software in CUDA or OpenCL, or in the intermediate language PTX, allows the software to be easily adapted to new hardware generations: in principle one should be able to simply recompile the software for a new GPU. However, this convenience does not seem to be compatible with our goal of high performance. Most of the GPU clusters that we have access to have the same Tesla architecture, increasing the value of optimizations targeting that architecture.

Since the PTX instruction definition is not very far from the actual instruction set of the Tesla-architecture GPUs, one could reasonably hope that using register variables in PTX followed by assembling to binary by `ptxas` from the NVIDIA toolchain would give the necessary control over the resources of a GPU. Unfortunately, we encountered major drawbacks of `ptxas`, including bad run-time performance, excessive compilation times, and sometimes complete failures to produce binary code.

A critical problem: Apparently the register allocator of `ptxas` was not designed for large kernels; the intended target applications are graphics shaders and small computing kernels. When the kernel is large, the allocation "leaks" (allocates too many) registers even when we explicitly generate code to be runnable within a smaller number of registers. Execution is often drastically slowed down when the compiler spills values to "local memory"; spilling values to "shared memory" is often even worse, preventing us from launching as many threads as we would like.

It is possible to code around the deficiencies of `ptxas`. However, the cost is significant in run time *and* programmer time. In our initial implementation work, we were forced after several months and much pulling of hair to employ a simple "schoolbook" method of implementing our critical multiplication subroutine, instead of any of the more advanced methods such as those mentioned in [1]. The end result was an overall performance hit of around 50%.

In many other cases the only workaround is to split the computation into several kernels which are executed one after another. This introduces additional overhead for kernel invocation. An ideal program would need to have as large kernels as possible to be handled by `ptxas` to avoid as much invocation overhead as possible. This would require much effort and experimentation on ideal kernel sizes.

We take a different approach, replacing `ptxas` by a tool that is able to handle large kernels without penalty. The next section introduces our new `qhasm-cudasm` tool. Beyond the basic benefit of supporting large kernels, `qhasm-cudasm` introduces a new input language that gives the programmer much more control over the hardware than PTX does, while at the same time achieving higher readability.

IV. `cudasm`, `qhasm`, AND `qhasm-cudasm`

This section describes the existing `cudasm` and `qhasm` assembly-language tools, and our new `qhasm-cudasm` assembly language.

A. `cudasm`

In 2007 van der Laan reverse-engineered the machine language of the NVIDIA GPUs. He released a `decuda` disassembler, translating each machine instruction into a readable format (somewhat similar to NVIDIA's documented PTX format, although the machine language turned out to be more complicated than PTX). Shortly afterwards he released a `cudasm` assembler. See [14].

Our experience is that `decuda` is by far the easiest way to figure out what `ptxas` is doing wrong. Anecdotal evidence suggests that `decuda` became moderately popular among serious programmers for exactly this reason. However, `cudasm` attracted far less attention. One paper [8] reported a successful application of `decuda` and `cudasm` to manually rewrite a small section of `ptxas` output, but said that this was "tedious" and hampered by `cudasm` bugs: "we must extract minimum region of binary code needed to be modified and keep remaining binary code unchanged ... implementation of `cudasm` is not entirely complete, it is not a good idea to write whole assembly manually and rely on `cudasm`."

We fixed various bugs in `cudasm`: for example, we found that memory offsets were sometimes silently ignored. Our fixed version of `cudasm` is capable of generating a fully functional 90000-GPU-instruction kernel for our software, and in fact is exactly what we use to generate all of the kernels that we are now running. However, *writing* these kernels in the `cudasm` input language would have been an extremely time-consuming job; we actually wrote our software in a new language, as discussed below.

B. `qhasm`

Years ago, one of the authors of this paper (Bernstein) tried very hard to convince `gcc` to emit his desired sequence of floating-point instructions for a performance-critical cryptographic application on an x86 CPU. Unfortunately, the x86 architecture had (at the time) only 8 floating-point registers; `gcc` expected to keep one of those registers in reserve for stack management; and trying to fit this particular application into 7 registers, rather than 8, seemed to irreparably compromise performance. He resorted to writing the same function in x86 assembly language; this required manually maintaining a chart of the floating-point values in each register, and redoing the chart several times to accommodate changes in the code.

The same author subsequently developed a higher-level assembly language, `qhasm`, to give him the same control as a traditional x86 assembly language with far less programming time. Many cryptographic speed records were set by software written in `qhasm`; see, e.g., [2], [6], and [9].

There are several differences between `qhasm` and a traditional assembly language. The most visible difference is the choice of syntax: `qhasm` uses readable C-like syntax such as `d = c + 3`, while a traditional assembler might use `lea 3(%ecx),%edx`. The most important difference, however, is that `qhasm` includes a very fast, very smart register allocator, mapping an unlimited stream of programmer-selected names to the small number of registers provided by a CPU. The programmer still controls the selection of instructions, still controls the order of instructions, and still controls *which* values remain in registers, but `qhasm` handles the tedious task of assigning registers. The programmer *can* easily specify a register assignment but almost never has to.

C. `qhasm-cudasm`

To bring the same level of assembly-language usability to GPUs we have built a new `qhasm-cudasm` language, reusing the `qhasm` register allocator to generate code that can be fed as input to our fixed version of `cudasm`. We wrote our new ECC2K-130 software in `qhasm-cudasm`; see subsequent sections of this paper for details.

The core of `qhasm-cudasm` is a GPU machine-description file, currently 2818 lines; many of those lines are automatically generated from a shorter script. What follows is a typical line from that file, split into five lines here for readability:

```
r = y + t:
>r=low32:
<y=low32:
<t=low32:
asm/add.b32 >r, <y, <t:
```

This line says that if `r` and `y` and `t` have been declared to be `low32` registers then the `qhasm-cudasm` instruction `r = y + t` reads `y`, reads `t`, writes `r` (eliminating any previous value), and corresponds to the `cudasm` instruction `add.b32 r, y, t`. Another line (with dots here for brevity) specifies the set of 64 `low32` registers:

```
:name:low32:$r0:$r1:...:$r63:
```

The reader might be wondering how each thread can have access to nearly 128 registers (as mentioned earlier) if there are only 64 registers listed here. The answer is that there are 60 extra `high32` registers:

```
:name:high32:$r64:$r65:...:$r123:
```

These are distinguished in `qhasm-cudasm` because they are distinguished in the GPU machine language: typical instructions can use `high32` in the first operand, or in the second operand, but not in the third operand.

We developed the machine-description file at the same time as writing code for the ECC2K-130 computation. We often changed syntax to improve readability or to avoid common error patterns. We added new instructions whenever we needed them:

```
r = s[p+m] if e signed<:
<e=cond:
inplace>r=high32:
<p=offset:
<r=high32:
#m:
asm/@<e.lt mov.b32 <r, s[<p+#m]:
```

This example is a predicated 32-bit load from shared memory into a `high32` register. The output value `r` can depend on the input value `r`, and the user can rely on it being assigned to the same register; this is what `inplace>r=high32` accomplishes. `#m` indicates an immediate constant. The register changes to the contents of the load if the `signed<` bit is set in the `e` predicate register.

We added an extra layer of preprocessing in front of `qhasm-cudasm`, using Ward's `m5` macro preprocessor [15]. An `m5` script is, except for some syntactic sugar, an `awk` program that prints another program; each of our `m5` scripts is a program that prints a `qhasm-cudasm` program.

We are using `qhasm-cudasm` for new applications, and are continuing to add instructions to the machine-description file. We are also building scripts to automate larger portions of the generation of the machine-description file.

D. Engineering a new implementation

For a typical function in the ECC2K-130 computation, such as `add` (described in the next section), we wrote a series of three implementations. The first implementation consisted of

- a simple C++ implementation `add.cpp` of a `paralleladd` function operating on data in CPU memory; and
- a test driver `addtest.cpp` calling `paralleladd`.

The second implementation consisted of

- CUDA code for `sharedadd` in `add.cu`, operating on data in shared memory;
- CUDA code for `kerneladd` in `kadd.cu`, operating on data in global memory by copying from global memory to shared memory and calling `sharedadd`;
- CUDA code for `paralleladd`, also in `kadd.cu`, operating on data in CPU memory by copying from the CPU to the GPU and then calling `kerneladd`; and
- the same test driver `addtest.cpp`.

The third implementation consisted of

- `qhasm-cudasm` code for `sharedadd` in `add.mq`;
- `qhasm-cudasm` code for `kerneladd` in `kadd.mq`, inlining `sharedadd` by including `add.mq`;
- the same CUDA code for `paralleladd` in `kadd.cu`, calling `kerneladd`; and
- the same test driver `addtest.cpp`.

Testing each version with the same test driver allowed typical bugs to be caught quickly.

It is common practice in GPU programming to implement functions twice, once in C and once in CUDA, with the same test driver; it is common practice in assembly-language programming to implement functions twice, once in C and once in assembly, with the same test driver. Our split between the first and second implementations followed the first practice, and our split between the second and third implementations imitated the second practice.

While working on the `qhasm-cudasm` versions we wrote a "big-kernel" implementation of the ECC2K-130 main loop. We wrote this main loop as a series of operations such as

```
xshift = r^2
xshift += r
d = global_Nd[j]
xshift += d
multprep r
```

in an ad-hoc language (not to be confused with `qhasm-cudasm`). We wrote a translator that converted this ad-hoc language to CUDA, replacing (e.g.) `xshift += r` with an appropriate call to the `sharedadd` function defined in `add.cu`, and replacing `d = global_Nd[j]` with an appropriate copy from global memory to shared memory. This CUDA implementation was extremely slow, and took an extremely long time for NVIDIA's tools to compile, but allowed the main loop to be tested independently of `qhasm-cudasm`.

We then wrote a translator that converted the same main loop to `qhasm-cudasm` code, automatically inlining individual `qhasm-cudasm` functions such as the `sharedadd` function defined in `add.mq`. Some data-flow misdeclarations had slipped past the individual tests and forced too many registers to be allocated in the main loop; in retrospect this could have been caught earlier by an extension to `qhasm-cudasm`, but in any case it was easy to diagnose and fix. This implementation took much less time to compile than the CUDA version and was much faster.

Afterwards we focused on optimizing various functions, producing the details described in subsequent sections. The third implementation described above was still in place, so any changes in `add.mq` were automatically tested by `addtest.cpp`. We also set up further scaffolding to measure time spent in various parts of the software, guiding our subsequent optimizations.

V. SCHEDULING INSTRUCTIONS: `add` AND `cadd`

The simplest arithmetic operation in the ECC2K-130 computation is an addition in a field of size 2^{131}: in other words, a XOR of two 131-bit input vectors, producing a 131-bit output vector. In our software this operation is called `add`. A "conditional" variant of the same operation, `cadd`, has an extra input bit that masks the second input vector: whenever a mask bit is 0, `cadd` simply copies the corresponding input bit from the first input, ignoring the second input.

As discussed earlier, we actually perform 128 independent computations in parallel. The `add` function actually takes two 131×128-bit input matrices (each stored in bitsliced row-major form as 131 consecutive 128-bit vectors), and produces an output matrix of the same size. The `cadd` function actually takes an extra 128-bit vector that masks the second input matrix.

This section describes our implementations of `add` and `cadd`. These functions are only small parts of the ECC2K-130 computation, but they are a useful starting point to illustrate the capabilities of `qhasm-cudasm`.

A. Predicate registers

Our 128 threads handle a vector of $131 \times 128 = 524 \times 32$ bits as 128×32 bits, then 128×32 bits, then 128×32 bits, then 128×32 bits, and finally 12×32 bits.

For the last 12×32 output bits, only 12 of the 128 threads are active. Each thread compares its "thread ID" (between 0 and 127) to the constant 12, and temporarily deactivates itself unless the thread ID is smaller than 12.

The GPU hardware has four special "predicate registers" in each thread (in addition to other registers) to store the results of such comparisons. Our ECC2K-130 software sets up one of these registers as follows:

```
  cond tid12
  tid12 tid - $(const(12))
```

A typical GPU instruction allows one input from "constant memory"; `qhasm-cudasm` converts `$(const(12))` into a constant-memory location, and arranges for that location to contain the integer 12. The line `cond tid12` declares `tid12` as a predicate register; `tid12 tid - $(const(12))` compares `tid` to 12 and puts the result of the comparison into `tid12`.

To save time inside functions such as `add` we perform this particular comparison once in the caller, rather than performing it again each time the function is called. This means that one of the four predicate registers is reserved long-term for `tid12`, but `tid12` is used frequently enough to justify this.

B. Shared-memory offsets

Each GPU core has 16384 bytes of "shared memory." A typical GPU instruction allows one input from shared memory. Reading shared memory is *often* as fast as reading a register, although it can trigger some additional bottlenecks. Shared memory has two advantages over registers: first, it allows threads to quickly communicate with each other; second, shared-memory indices can be variables, while register numbers are always constant.

The GPU hardware has four special "offset registers" in each thread to store indices into shared memory. In our ECC2K-130 software the caller sets up an offset register before calling `add`, `cadd`, etc.:

```
  offset tid4off
  tid4off = tid << 2
```

Here `offset tid4off` declares `tid4off` to be one of the offset registers, and `tid4off = tid << 2` performs a shift by 2, i.e., a multiplication by 4. Note that a shift of a general-purpose register can put its output into an offset register.

C. How the `add` *function works*

Our `m5` function `add(to,from)` prints code that reads two inputs from shared memory, XORs the inputs, and writes the output on top of the first input. The input addresses are `to` and `from`; the output address is `to`. This code is not a machine-level function requiring `call` and `ret` instructions; it is inlined into the code generated by the caller, just like a macro expansion.

The `add` function begins by loading the first input:

```
syncthreads
new @x4
@x0 = s[tid4off + $to]
@x1 = s[tid4off + $(to + 512)]
@x2 = s[tid4off + $(to + 1024)]
@x3 = s[tid4off + $(to + 1536)]
@x4 = s[tid4off + $(to + 2048)] if tid12 signed<
```

Here `$` starts a compile-time computation; if `to` happens to be 4000, for example, then the fourth line above is converted into `@x1 = s[tid4off + 4512]`.

Recall that the `tid12` predicate register compared the thread ID to 12. The predicate `if tid12 signed<` skips the instruction on the same line unless the thread ID is smaller than 12. Note that each predicate register actually has four different comparison bits, allowing a variety of different predicates: `signed<`, `signed<=`, etc.

Two aspects of the above code help the `qhasm` register allocator manage data flow. First, `@` creates (at the `m5` level) a register name specific to this function call, avoiding accidental data flow between function calls. Second, `new @x4` informs the register allocator that there is no data flow from any previous use of the `@x4` register. This is not necessary for `@x0`, `@x1`, `@x2`, `@x3`: a register written by a *non-predicated* assignment has value independent of its previous value.

The `add` function continues by loading the second input:

```
new @y4
@y0 = s[tid4off + $from]
@y1 = s[tid4off + $(from + 512)]
@y2 = s[tid4off + $(from + 1024)]
@y3 = s[tid4off + $(from + 1536)]
@y4 = s[tid4off + $(from + 2048)] if tid12 signed<
```

The function then computes and stores the results:

```
@x0 ^= @y0
@x1 ^= @y1
@x2 ^= @y2
@x3 ^= @y3
@x4 ^= @y4
```

```
s[tid4off + $to] = @x0
s[tid4off + $(to + 512)] = @x1
s[tid4off + $(to + 1024)] = @x2
s[tid4off + $(to + 1536)] = @x3
s[tid4off + $(to + 2048)] = @x4 if tid12 signed<
```

The cadd function is similar but includes two extra instructions to load the mask and five extra AND instructions.

One can sometimes merge a load instruction and an XOR instruction into a single load-and-XOR instruction. However, the load-and-XOR instruction requires the immediate value used in the address computation to be in the range from -128 to 127, while the immediate values in the code shown above are almost never in this range. We subsequently experimented with rearranging the responsibilities of threads, assigning the first thread to the first 5×32 bits, the second thread to the next 5×32 bits, etc., so that each load address would be just 4 bytes after the previous rather than 512 bytes after the previous. This saved five instructions but cost four instructions to initialize two new offset registers; competition for offset registers meant that we could not keep these two offset registers longer than the function call. We plan to experiment further with eliminating this cost by more comprehensively rearranging our data structures, interleaving several 131×128-bit matrices with each other.

D. Performance

One can crudely model a GPU core as following 8 instructions per cycle. This model suggests that 128 threads would follow the 21 instructions shown above (10 loads, 5 stores, 5 XORs, 1 synchronization) in 336 cycles, and would follow the 28 cadd instructions in 448 cycles.

The GPU has a cycle counter. This counter is labelled as halfclock in qhasm-cudasm, because it actually counts once every two cycles. The cycle counter shows that the cadd instructions *plus* the cycle-counting time actually take 644 cycles, while our CUDA implementation of the same function takes 1106 cycles. An inspection of the machine-language code produced by nvcc and ptxas shows several sources of inefficiency, such as excessive use of offset registers. We could try to tweak our CUDA code, hoping for better output from nvcc and ptxas, but writing the code in qhasm-cudasm is less effort.

VI. HANDLING MEMORY CONFLICTS: sq AND csq

The ECC2K-130 computation involves 8 different sq operations. Each of these operations takes as input one 131×128 bit matrix and applies a fixed permutation to the rows of the input matrix to produce again a 131×128 bit matrix as output. What is different in these 8 sq operations is the permutation applied to the rows. Each matrix row is stored in 4 successive 32-bit integers, so that one sq operation requires $4 \cdot 131 = 524$ load instructions and another 524 store instructions.

This sq operation takes 5 load instructions and 5 store instructions issued to 128 threads; here two instructions (one load and one store) are conditional depending on the thread ID. However, the performance of 128 concurrent load or store operations performed by 128 threads is highly dependent on the addresses of the data loaded or stored.

Shared memory is organized into 16 “banks” of memory: 16 consecutive 32-bit words are spread across the banks, one per bank. The 128 threads are organized into 4 “warps”; each warp is divided into 2 “half-warps”. The 16 threads of one half-warp can execute a load operation of the sq operation in one cycle only if they all load from different memory banks. If two or more threads within the same half-warp load from or store to different addresses on the same memory bank in the same instruction, these requests are serialized.

We first group every four adjacent threads and let them operate on the four 32-bit words of one matrix row. Four such groups form a half-warp. We try to assign the 131 rows to such groups in a way that avoids all memory-bank conflicts.

We use a lookup table in constant memory to assign the 131 rows to 32 thread groups. Each entry of this lookup table requires only one byte; we pack 4 lookup-table entries into one 32-bit value.

The memory-bank restrictions also hold for storing the data, so avoiding memory-conflicts only for loading may yield bank conflicts when storing the data at the locations given by the fixed row permutation. We implemented a tool that uses a greedy approach to compute two lookup tables, one for loading and one for storing, that avoid almost all bank conflicts. These tables are linked through the permutation given by the sq operation.

Four of the sq operations are used very frequently, so we load the packed 20 load and 20 store positions from constant memory once at the beginning of the computation and then keep them in long-term registers:

```
sqseq_tmp = tid uint32>> 2
sqseq_pos = sqseq_tmp << 2

low32 sqseq_in0
low32 sqseq_in1
low32 sqseq_in2
low32 sqseq_in3
low32 sqseq_in4
sqseq_in0 = c0[sqseq_pos + 0]
sqseq_in1 = c0[sqseq_pos + 128]
sqseq_in2 = c0[sqseq_pos + 256]
sqseq_in3 = c0[sqseq_pos + 384]
sqseq_in4 = c0[sqseq_pos + 512]

low32 sqseq_out0
low32 sqseq_out1
low32 sqseq_out2
low32 sqseq_out3
low32 sqseq_out4
sqseq_out0 = c0[sqseq_pos + $(0+1048)]
sqseq_out1 = c0[sqseq_pos + $(128+1048)]
sqseq_out2 = c0[sqseq_pos + $(256+1048)]
sqseq_out3 = c0[sqseq_pos + $(384+1048)]
sqseq_out4 = c0[sqseq_pos + $(512+1048)]
```

Extracting one of the 4 packed positions from a 32-bit integer requires (at most) one shift and one mask instruction. For example, the following code extracts one position from `sqseq_out0`:

```
@p0 = sqseq_out0 uint32>> 4
@p0 &= $(const(4080))
```

Similar comments apply to `csq`, a conditional version of `sq`.

VII. Reducing serialization: `hamming`

The `hamming` operation takes as input a 131 × 128 bit matrix, computes the Hamming weight (sum of bits) for each column of the matrix, and stores the binary representation of the Hamming weight in the first bits of each column.

Adding up all bits *in order* is a completely serial process, but we obtain some parallelization by changing the order of additions. The basic operation takes three input vectors `a`, `b`, and `c` and computes two output vectors `bot` and `top` so that for each position i it holds that $\mathtt{a[i]}+\mathtt{b[i]}+\mathtt{c[i]}=\mathtt{top[i]}\cdot 2+\mathtt{bot[i]}$. It is possible to schedule 32 such operations in parallel under some conditions on the addresses modulo 16. This means that in the first step a total of $3\times 32=96$ vectors can be handled, leading to 32 resulting vectors on level 1 (`top`) and $32+(131-96)=67$ results on level 0 (`bot`). In the next step results on level 2 can be computed by using vectors on level 1 as inputs; we carefully arrange vector positions so that in the same instruction results on level 1 and 0 are computed.

The computation is composed of two routines, FullAdder(`x,in0,in1,in2,out0,out1,threads`) and HalfAdder(`x,in0,in1,out0,out1,threads`). The core of the FullAdder routine consists of the following code:

```
syncthreads
@todo tid - $(const(4 * threads))
new @a
new @b
new @c
@a=s[tid4off + $(x + 16*in0)] if @todo signed<
@b=s[tid4off + $(x + 16*in1)] if @todo signed<
@c=s[tid4off + $(x + 16*in2)] if @todo signed<
@sum=@a ^ @b
@aandb=@a & @b
@candsum=@c & @sum
@bot=@sum ^ @c
@top=@aandb ^ @candsum
s[tid4off + $(x + 16*out0)]=@bot if @todo signed<
s[tid4off + $(x + 16*out1)]=@top if @todo signed<
```

The core of the HalfAdder routine consists of the following code:

```
syncthreads
@todo tid - $(const(4 * threads))
new @a
new @b
@a=s[tid4off + $(x + 16*in0)] if @todo signed<
@b=s[tid4off + $(x + 16*in1)] if @todo signed<
@bot=@a ^ @b
@top=@a & @b
s[tid4off + $(x + 16*out0)]=@bot if @todo signed<
s[tid4off + $(x + 16*out1)]=@top if @todo signed<
```

Overall we use 19 FullAdder stages and 2 HalfAdder stages.

VIII. Batching operations: `multprep` etc.

The `multprep` operation takes as input a 131 × 128 bit matrix and transforms this matrix in place using a particular pattern of XORs. We first explain the transformation as a series of operations on 128-bit vector variables `c0`, …, `c130` holding the matrix rows, and then discuss parallelization of the transformation.

The transformation starts with the short initial computation

```
c126^=c128
c125^=c129
c124^=c130
```

and continues with 6 levels of conversion. Level 1 consists of one computation involving 126 rows:

```
c62^=c64
c61^=c65
c60^=c66
...
c0^=c126
```

Level 2 consists of 2 computations, each involving 62 rows:

```
c30^=c32
c29^=c33
...
c0^=c62
```

and

```
c94^=c96
c93^=c97
...
c64^=c126
```

Level 3 consists of 4 computations, each involving 30 rows; level 4 of 8 computations, each involving 14 rows; level 5 of 16 computations, each involving 6 rows; and level 6 of 32 computations, each involving 2 rows. The basic structure of each of these computations on each level is XORing the upper rows into the lower rows as shown for levels 1 and 2.

We merge levels 1 and 2 of conversion, and then assign the resulting 125 computations to 128 threads as follows. As in previous sections, each group of four adjacent threads operates on the four 32-bit integers of one matrix row. We assign the first group (threads 0, 1, 2, 3) to the operations in levels 1 and 2 on `c0`, `c62`, `c64`, and `c126`:

```
c62^=c64
c0^=c126
c64^=c126
c0^=c62
```

The actual qhasm-cudasm code consists of the following 11 instructions, where x is the starting position of the input matrix in shared memory:

```
@a=s[tid4off + $(x)]
@f=s[@zoff + $(x+4*4*(62+64))]
@b=s[tid4off + $(x+4*4*64)]
@c=s[@zoff + $(x+4*4*62)]
@a^=@f
@c^=@b
@a^=@c
@b^=@f
s[tid4off + $(x)]=@a
s[tid4off + $(x+4*4*64)]=@b if @check signed<
s[@zoff + $(x+4*4*62)]=@c
```

Thread 1 performs the same operations on c1, c61, c65 and c125, and so on; thread 30 performs the operations on c30, c32, c94 and c96. In a similar way we merge levels 3 and 4 and levels 5 and 6.

Merging two consecutive levels keeps most threads busy during the whole computation, and results in only a small number of memory-bank conflicts for 128 threads working on the whole matrix.

IX. Minimizing code size: ppn and ppp

The most time-consuming operation in our software is field multiplication. This consists of a 131 × 131 → 261-bit polynomial multiplication, followed by a 261 → 131-bit reduction. As mentioned earlier, field multiplication has two flavors, ppn and ppp; these have the same polynomial multiplication but differ in the details of the reduction.

Our polynomial-multiplication code consists of 725 instructions and uses almost all of the 16KB shared memory. See [4, Section 5] for details of our multiplication strategy. Reduction is similar to the multprep operation described in Section VIII but is about twice as long.

Our main loop originally consisted of straight-line code for a batch of B iterations, using $5B+5$ multiplications. We wanted B to be reasonably large, at least 32, so as not to notice the $+5$ overhead here, but we ran into a performance problem: 165 straight-line multiplications do not fit into the second-level GPU instruction cache. The cores cannot load instructions from global memory quickly enough to keep the ALUs busy.

To avoid this problem we reduced code size by the following compression strategy: identify a large, contiguous, frequently used part of the code, and convert it into a machine-level function. Our code uses Tesla instructions call.label and return to enter and leave the function; these instructions manage return addresses in hardware.

We are departing here from the standard practice on GPUs, namely to inline all code. There is no common calling convention; saving registers to a stack in global memory would be highly inefficient. We avoid register spills, and circumvent the introduction of a calling convention, by instructing the register allocator to find a fixed register assignment suitable for all calls to a particular function.

The largest contiguous section of code in our kernel, and also the largest consumer of time, is the 131-bit polynomial multiplication. We put just one copy of the code into the kernel, labeled as mult_131x131. All m5-level calls to the multiplication—which would normally cause inlining in the assembly code—are replaced by machine-level calls to mult_131x131.

This mechanism for function calls goes beyond any previous use of qhasm. It increases pressure on the register allocator and requires us to place careful hints to the register allocator about expired register variables. But the code works, and most of its run time is spent on instructions that fit into the instruction cache.

We also arranged large parts of the computation into size-B loops, but the most natural way to do this still contained 16 multiplications. Function calls are more flexible than loops.

We further reduced code size by using *half instructions*. Most instructions are encoded in 64-bit words, but some simple instructions can be encoded in 32-bit words:

```
shortinsn @p0 += @slicex
shortinsn @p1 += @slicex
shortinsn @p2 += @slicex
shortinsn @p3 += @slicex
```

A pair of these half instructions fills up one slot of a "regular" full instruction.

X. Instruction scheduling

NVIDIA's documentation does not suggest any real importance to the order of instructions. There are warnings regarding large latencies for global-memory access, but there is no documentation of the latencies of any other instructions, or of any other interactions between nearby instructions.

We nevertheless found two ways to save time by reordering instructions. First, given dependent instructions such as

```
@c ^= @b
@a ^= @c
```

we moved another instruction in between. If the first instruction is issued to 128 threads then it will occupy the 8 ALUs in the core for only 16 cycles, but apparently it has a latency of more than 16 cycles, stalling the next instruction if there is a dependency.

Second, we tried to avoid adjacent shared-memory accesses. There seems to be a latency of more than 16 cycles from one shared-memory instruction to the next shared-memory instruction, even if the instructions are not dependent.

After this round of optimization we found that our qhasm-cudasm code was running at 63 million iterations per second on a GTX 295, more than 4 trillion useful bit operations per second. This is only a quarter of the maximum theoretical capacity of a GTX 295 but it is more than twice as fast as our best code compiled by nvcc and ptxas.

XI. Conclusion

In this paper, we have described a high-level assembly-language toolchain, called qhasm-cudasm, that simultaneously allows efficient programming and complete control over raw GPU hardware. qhasm-cudasm programmers have full control over instruction selection and scheduling, as well as register allocation and spills. Furthermore, qhasm-cudasm has automated the most tedious task of register assignment while giving the programmer the maximum degree of freedom in software architectural exploration, allowing creative programming to reach new heights of performance on the GPU. This allowed us to build a 90000-machine-instruction kernel for the largest cryptanalysis project in history, and a streamlined kernel outperforming the best nvcc implementation by 148%. We expect to be able to use the qhasm-cudasm toolchain to set speed records for many other applications, and in fact have already used it to almost double the speed of a computation in quantum chemistry.

There are already several projects that aim to provide analogous features to cudasm for the latest GPUs from NVIDIA and AMD: asfermi [10] targets NVIDIA's Fermi architecture; amdasm [13] targets AMD GPUs; calasm [12], from one of the authors of this paper (Niederhagen), also targets AMD GPUs. Future versions of our software will build upon these projects, bringing the usability of qhasm-cudasm to all of these GPUs.

References

[1] Daniel V. Bailey, Lejla Batina, Daniel J. Bernstein, Peter Birkner, Joppe W. Bos, Hsieh-Chung Chen, Chen-Mou Cheng, Gauthier van Damme, Giacomo de Meulenaer, Luis Julian Dominguez Perez, Junfeng Fan, Tim Güneysu, Frank Gurkaynak, Thorsten Kleinjung, Tanja Lange, Nele Mentens, Ruben Niederhagen, Christof Paar, Francesco Regazzoni, Peter Schwabe, Leif Uhsadel, Anthony Van Herrewege, and Bo-Yin Yang. Breaking ECC2K-130. Cryptology ePrint Archive, Report 2009/541, 2009. http://eprint.iacr.org/2009/541.

[2] Daniel J. Bernstein. Curve25519: new Diffie-Hellman speed records. In *Public Key Cryptography—PKC 2006*, volume 3958 of *Lecture Notes in Computer Science*, pages 207–228. Springer, 2006. http://cr.yp.to/papers.html#curve25519.

[3] Daniel J. Bernstein. qhasm software package, 2007. http://cr.yp.to/qhasm.html.

[4] Daniel J. Bernstein, Hsieh-Chung Chen, Chen-Mou Cheng, Tanja Lange, Ruben Niederhagen, Peter Schwabe, and Bo-Yin Yang. ECC2K-130 on NVIDIA GPUs. In *Progress in Cryptology—INDOCRYPT 2010*, volume 6498 of *LNCS*, pages 328–346. Springer, 2010. http://eprint.iacr.org/2012/002/.

[5] Daniel J. Bernstein and Tanja Lange. Type-II optimal polynomial bases. In *Arithmetic of Finite Fields—WAIFI 2010*, volume 6087 of *Lecture Notes in Computer Science*, pages 41–61. Springer, 2010. http://eprint.iacr.org/2010/069.

[6] Daniel J. Bernstein and Peter Schwabe. New AES software speed records. In *Progress in Cryptology—INDOCRYPT 2008*, volume 5365 of *Lecture Notes in Computer Science*, pages 322–336. Springer, 2008. http://eprint.iacr.org/2008/381.

[7] Certicom. Certicom ECC challenge. http://www.certicom.com/images/pdfs/cert_ecc_challenge.pdf, 1997.

[8] Lung-Sheng Chien. Hand-tuned SGEMM on GT200 GPU. http://oz.nthu.edu.tw/~d947207/NVIDIA/SGEMM/HandTunedSgemm_2010_v1.1.pdf, 2010.

[9] Neil Costigan and Peter Schwabe. Fast elliptic-curve cryptography on the Cell Broadband Engine. In *Progress in Cryptology—AFRICACRYPT 2009*, volume 5580 of *Lecture Notes in Computer Science*, pages 368–385. Springer, 2009. http://eprint.iacr.org/2009/016.

[10] Yunqing Hou. asfermi: An assembler for the NVIDIA Fermi instruction set, 2011. http://code.google.com/p/asfermi/, accessed Nov. 1, 2011.

[11] Thorsten Kleinjung, Kazumaro Aoki, Jens Franke, Arjen K. Lenstra, Emmanuel Thomé, Joppe W. Bos, Pierrick Gaudry, Alexander Kruppa, Peter L. Montgomery, Dag Arne Osvik, Herman J. J. te Riele, Andrey Timofeev, and Paul Zimmermann. Factorization of a 768-bit RSA modulus. In Tal Rabin, editor, *CRYPTO*, volume 6223 of *Lecture Notes in Computer Science*, pages 333–350. Springer, 2010.

[12] Ruben Niederhagen. Calasm, 2011. http://www.polycephaly.org/projects/calasm/.

[13] Ádám Rák. AMD-GPU-Asm-Disasm, 2011. https://github.com/rakadam/AMD-GPU-Asm-Disasm/, accessed Nov. 1, 2011.

[14] Wladimir J. van der Laan. Cubin utilities. http://wiki.github.com/laanwj/decuda/, 2007.

[15] William A. Ward, Jr. Algorithm 803: a simpler macro processor. *ACM Transactions on Mathematical Software*, 26:310–319, 2000.