S6349 - XMP LIBRARY INTERNALS

Niall Emmart
University of Massachusetts

Follow on to S6151 – XMP: An NVIDIA CUDA – Accelerated Big Integer Library
High Performance Modular Exponentiation

\[ A^K \mod P \]

Where A, K and P are hundreds to thousands of bits
Applications

Modular exponentiation is a key cryptographic primitive, used in:

• RSA
• Diffie-Hellman
• Digital signature algorithms
• Prime generation
• Factorization

Several applications require high volume modular exponentiation:

• SSL connection negotiation, i.e., secure web servers at large data centers
• Packet authentication for new network designs, such as Content Centric Networking
• Code breaking
Achieving High Performance

- Traditional fixed radix number system
- One problem instance per thread
- Keep as much on chip as possible
- Extensive PTX assembly. Use carry flag and special instructions, \texttt{imad32.x.cc} on Fermi and Kepler, use \texttt{xmad.x.cc} on Maxwell.
- Use fast algorithms: Montgomery reduction, fast squaring, Karatsuba (multiplication and squaring), fixed window exponentiation.
Algorithms
Fast Exponentiation

Observe that:

\[ A^{0x5364} = (((A^5)^{16} (A^3))^{16} (A^6))^{16} (A^4) \]

This leads to a natural algorithm – Fixed Window Exponentiation:

1) Precompute table, \( A^0, A^1 \ldots A^{15} \)
2) Process the exponent from most significant to least significant digit. At each step either square the current, or multiply the current by an entry from the window table.
3) Mod P distributes over multiplication, thus \( A^K \mod P \) can be computed by reducing \( \mod P \) after each multiplication step
The Fermi and Kepler multipliers are 32 bit by 32 bit multiplier. The basic form is:

```
 imad.x.{lo,hi}.cc d, a, b, c;
```

Computes a 64 bit product of \(a \times b\), selects the low or high 32 bits of the product and adds \(c\) with optional carry in and optional carry out. The result is assigned to \(d\).
# Multiplication on Kepler (IMAD)

<table>
<thead>
<tr>
<th></th>
<th>(A_3)</th>
<th>(A_2)</th>
<th>(A_1)</th>
<th>(A_0)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(B_3)</td>
<td>(B_2)</td>
<td>(B_1)</td>
<td>(B_0)</td>
</tr>
<tr>
<td></td>
<td>(L(A_3B_0))</td>
<td>(L(A_2B_0))</td>
<td>(L(A_1B_0))</td>
<td>(L(A_0B_0))</td>
</tr>
<tr>
<td>ADD</td>
<td>(H(A_3B_0))</td>
<td>(H(A_2B_0))</td>
<td>(H(A_1B_0))</td>
<td>(H(A_0B_0))</td>
</tr>
<tr>
<td>(H(A_3B_1))</td>
<td>(L(A_3B_1))</td>
<td>(L(A_2B_1))</td>
<td>(L(A_1B_1))</td>
<td>(L(A_0B_1))</td>
</tr>
<tr>
<td>ADD</td>
<td>(H(A_3B_2))</td>
<td>(L(A_3B_2))</td>
<td>(L(A_2B_2))</td>
<td>(L(A_1B_2))</td>
</tr>
<tr>
<td>(H(A_3B_3))</td>
<td>(L(A_3B_3))</td>
<td>(L(A_2B_3))</td>
<td>(L(A_1B_3))</td>
<td>(L(A_0B_3))</td>
</tr>
</tbody>
</table>

Use [madc.lo.cc](http://madc.lo.cc) and [madc.hi.cc](http://madc.hi.cc)
Multiplication on Maxwell (XMAD)

• On Maxwell, a 32-bit `madc.lo.cc` instruction takes 4 instructions.
• A 32-bit `madc.hi.cc` takes 6 instructions.

Thus the MP multiplication algorithms takes roughly $10 \times N^2$ Maxwell instructions!

We can do better!
The Maxwell multiplier is a 16 bit by 16 bit multiplier. The basic form is:

\[ \text{xm} \text{ad} \cdot \text{x} \cdot \text{cc} \ d, \ a \cdot \{h0|h1\}, \ b \cdot \{h0|h1\}, \ c; \]

It selects a half word from \( a \) and a half word from \( b \), computes the 32 bit full product and adds \( c \), with carry in and carry out.

Consider the simple case of computing \( A \times B \) where \( A \) and \( B \) are each 32 bits, to generate a 64 bit result:

\[
A \times B = \begin{array}{c}
\begin{array}{c}
AH \times BH \\
AL \times BL
\end{array}
\end{array}
\]

These two products are half word aligned.

It requires a lot of work to integrate the half word aligned products.
Multiplication on Maxwell (cont)

<table>
<thead>
<tr>
<th>$A_3$</th>
<th>$A_2$</th>
<th>$A_1$</th>
<th>$A_0$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Terms

Green terms are full word aligned

Red terms are half word aligned
Multiplication on Maxwell (cont)

SUM THE RED TERMS AND SHIFT LEFT 16 BITS USING PRMT

ADD IN THE GREEN TERMS

Roughly $4N^2$ instructions
Fast Squaring – exploit symmetry

<table>
<thead>
<tr>
<th></th>
<th>A_3</th>
<th>A_2</th>
<th>A_1</th>
<th>A_0</th>
</tr>
</thead>
<tbody>
<tr>
<td>H(A_3A_0)</td>
<td>L(A_3A_0)</td>
<td>L(A_2A_0)</td>
<td>L(A_1A_0)</td>
<td>L(A_0A_0)</td>
</tr>
<tr>
<td>L(A_3A_1)</td>
<td>H(A_2A_0)</td>
<td>H(A_1A_0)</td>
<td>H(A_0A_0)</td>
<td>L(A_0A_1)</td>
</tr>
<tr>
<td>H(A_3A_1)</td>
<td>H(A_2A_1)</td>
<td>H(A_1A_1)</td>
<td>H(A_0A_1)</td>
<td>L(A_0A_1)</td>
</tr>
<tr>
<td>L(A_3A_2)</td>
<td>L(A_2A_2)</td>
<td>L(A_1A_2)</td>
<td>L(A_0A_2)</td>
<td>L(A_0A_2)</td>
</tr>
<tr>
<td>H(A_3A_2)</td>
<td>H(A_2A_2)</td>
<td>H(A_1A_2)</td>
<td>H(A_0A_2)</td>
<td>L(A_0A_2)</td>
</tr>
<tr>
<td>L(A_3A_3)</td>
<td>L(A_2A_3)</td>
<td>L(A_1A_3)</td>
<td>L(A_0A_3)</td>
<td>L(A_0A_3)</td>
</tr>
<tr>
<td>H(A_3A_3)</td>
<td>H(A_2A_3)</td>
<td>H(A_1A_3)</td>
<td>H(A_0A_3)</td>
<td>L(A_0A_3)</td>
</tr>
</tbody>
</table>

Compute the Red values, double it and add in the grey diagonal values
Montgomery Reduction

- We use a variant called “Almost Montgomery”
- Replaces an expensive Mod P operation with a special type of multiply, a word shift and a subtract
- Run time is typically 10-20% more than a full multiply
- Requires special pre and post processing

See Wikipedia for more details and examples
Strategy for Small Sizes – Three N

Small sizes --- 128 to 512 bits

• Problem instance per thread
• Store three multiple precision values in register in each thread
• We call this the Three N model

Pros: Very efficient and compute intensive

Cons:
1. Need to be very careful with register usage to ensure enough warps to saturate the ALUs.
2. Doesn’t scale past 512 bits
Strategy for Small Sizes (cont)

Step 1
Multiply

A Value (N registers)

| A₀ | A₁ | A₂ | ... | A_{n-1} |

B Value (N registers)

| B₀ | B₁ | B₂ | ... | B_{n-1} |

Product (2N registers, overwrites B)

Step 2
Reduce

| A₀ | A₁ | A₂ | ... | A_{n-1} |

| AB₀ | AB₁ | AB₂ | ... | AB_{2n-1} |

Montgomery Reduction
Strategy for Large Sizes

Large sizes --- 1024 bits and up

1) For efficiency we want a problem instance per thread
2) Not enough registers to do Three N
3) MP values must be stored in global memory
Strategy for Large Sizes (cont)

We must consider memory access patterns:

<table>
<thead>
<tr>
<th></th>
<th>A₃ B₃</th>
<th>A₂ B₂</th>
<th>A₁ B₁</th>
<th>A₀ B₀</th>
</tr>
</thead>
<tbody>
<tr>
<td>L</td>
<td>L(A₃B₀)</td>
<td>L(A₂B₀)</td>
<td>L(A₁B₀)</td>
<td>L(A₀B₀)</td>
</tr>
<tr>
<td>H</td>
<td>H(A₃B₀)</td>
<td>H(A₂B₀)</td>
<td>H(A₁B₀)</td>
<td>H(A₀B₀)</td>
</tr>
<tr>
<td>L</td>
<td>L(A₃B₁)</td>
<td>L(A₂B₁)</td>
<td>L(A₁B₁)</td>
<td>L(A₀B₁)</td>
</tr>
<tr>
<td>H</td>
<td>H(A₃B₁)</td>
<td>H(A₂B₁)</td>
<td>H(A₁B₁)</td>
<td>H(A₀B₁)</td>
</tr>
<tr>
<td>L</td>
<td>L(A₃B₂)</td>
<td>L(A₂B₂)</td>
<td>L(A₁B₂)</td>
<td>L(A₀B₂)</td>
</tr>
<tr>
<td>H</td>
<td>H(A₃B₂)</td>
<td>H(A₂B₂)</td>
<td>H(A₁B₂)</td>
<td>H(A₀B₂)</td>
</tr>
<tr>
<td>L</td>
<td>L(A₃B₃)</td>
<td>L(A₂B₃)</td>
<td>L(A₁B₃)</td>
<td>L(A₀B₃)</td>
</tr>
<tr>
<td>H</td>
<td>H(A₃B₃)</td>
<td>H(A₂B₃)</td>
<td>H(A₁B₃)</td>
<td>H(A₀B₃)</td>
</tr>
</tbody>
</table>

Row Oriented:  N accesses across all of A
Column Oriented:  N accesses across all of A and all of B
Strategy for Large Sizes (cont)

These access patterns are OK on a CPU because of the large fast cache. On the GPU we only have a tiny amount of cache on a per thread basis.

Solution: Break A and B into fixed sized chunks or “digits” of, say, 256 bits:

<table>
<thead>
<tr>
<th>Digit 0</th>
<th>Digit 1</th>
<th>Digit 2</th>
<th>Digit 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>A=1024 Bits</td>
<td>Digit 0</td>
<td>Digit 1</td>
<td>Digit 2</td>
</tr>
<tr>
<td>B=1024 Bits</td>
<td>Digit 0</td>
<td>Digit 1</td>
<td>Digit 2</td>
</tr>
</tbody>
</table>
Strategy for Large Sizes (cont)

• It’s effective because loading is linear in the length of the digit, but multiplication, division and reduction are quadratic in the length of the digit.

• In theory you can adjust the digit size to achieve higher or lower compute to load/store ratios. In practice, there is more overhead if the digit size does not evenly divide the modulus size.

• Now you can re-derive all of the operations: add, sub, mul, div, montgomery reduce in terms of operation on digits.
Results and Compiler Support for XMAD
The E5-2698v3 is a 16 core Xeon at 2.3 GHz
Compiler support for XMAD

Unfortunately, there is no single PTX instruction that corresponds to a Maxwell XMAD. However, we can define a sequence of instructions:

```c
__device__ __forceinline__
void XMADLL(uint32& d, uint32& a, uint32& b, uint32& c) {
    asm volatile ("{
        .reg .u16 %al, %ah, %bl, %bh;
        mov.b32 {%al,%ah},%1;
        mov.b32 {%bl,%bh},%2;
        mul.wide.u16 %0, %al, %bl;
        add.u32 %0, %0, %3;
    }" : "=r"(d) : "r"(a), "r"(b), "r"(c));
}
```

As of CUDA 8, the compiler will recognize this (and related) sequences and convert all these instructions into a single XMAD on Maxwell.

A big shout out to the compiler team for all their help and excellent support of this library!
Thank you!

Questions?