Implementation of RSA 2048 on GPUs

Marcelo E. Kaihara
EPFL – LACAL
Nov. 4, 2010

Motivation

- NIST Recommendations for Key Management (SP 800-57)
- NIST DRAFT recommendation for the Transitioning of Cryptographic Algorithms and Key Sizes (SP 800-131)

RSA 1024

Deprecated from January 1, 2011

RSA 2048

8x Computational Effort

Object

- Use GPUs as cryptographic accelerators to offload work from the CPU.

- Low latency
- Generic implementation
- Server application

- Parallel implementation
- OpenCL
- Speed

RSA 2048 Decryption

Decryption

$z = c^d \bmod m \qquad m = p \cdot q \qquad e \cdot d = 1 \bmod \varphi(m)$

Precomputed values

$(p, q, dP, dQ, qInv)$

$dP = e^{-1} \bmod (p - 1)$

$dQ = e^{-1} \bmod (q - 1)$

$dInv = q^{-1} \bmod p$

Chinese Remainder Theorem

$z_1 = c^{dP} \bmod p$

$z_2 = c^{dQ} \bmod q$

$h = qInv \cdot (z_1 - z_2) \bmod p$

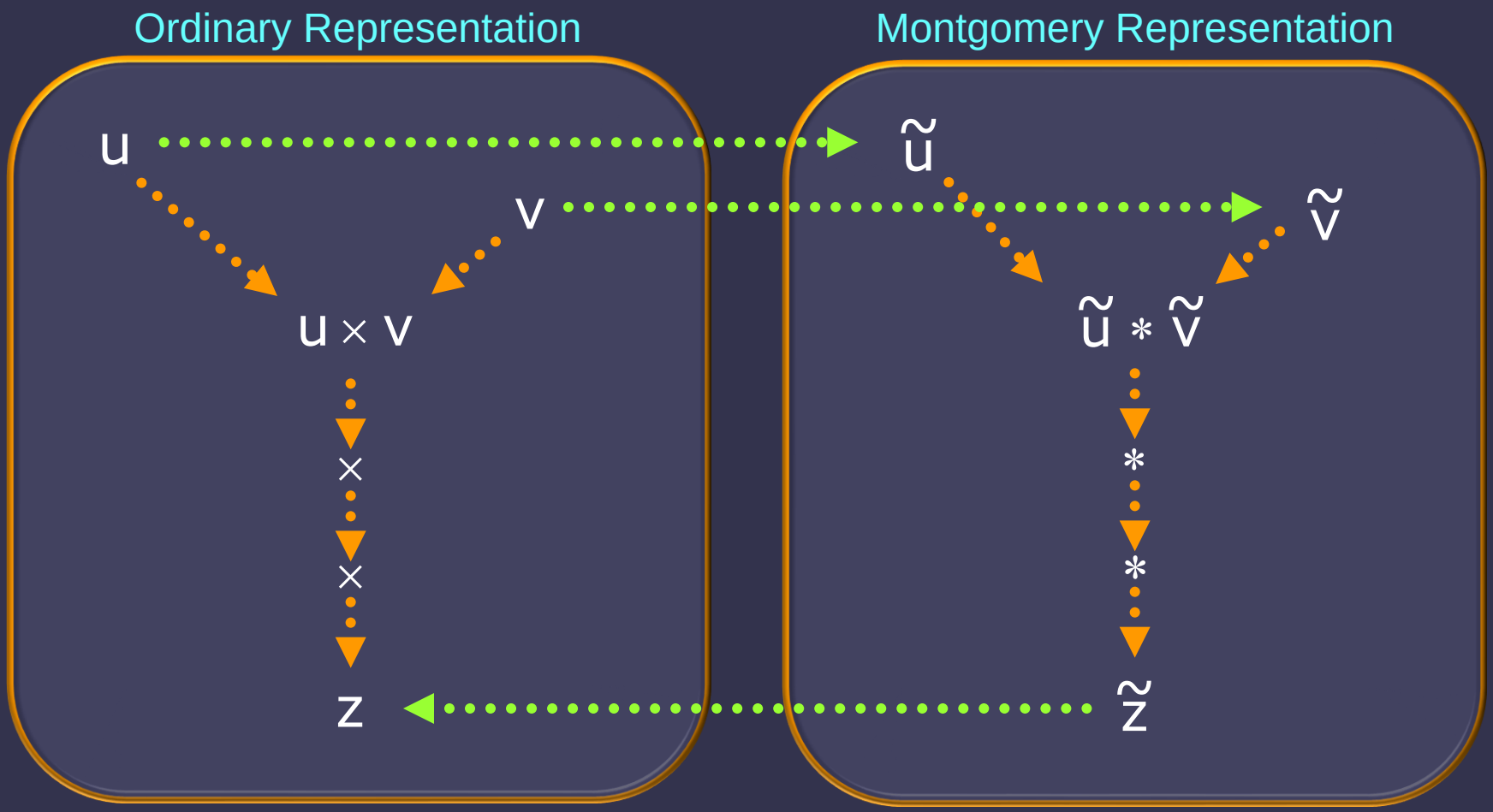
$z = z_2 + h \cdot q$

Mod Exp 1024 moduli
(32 limbs of 32-bits)

$s = 32 \qquad B = 2^{32}$

Montgomery Multiplication

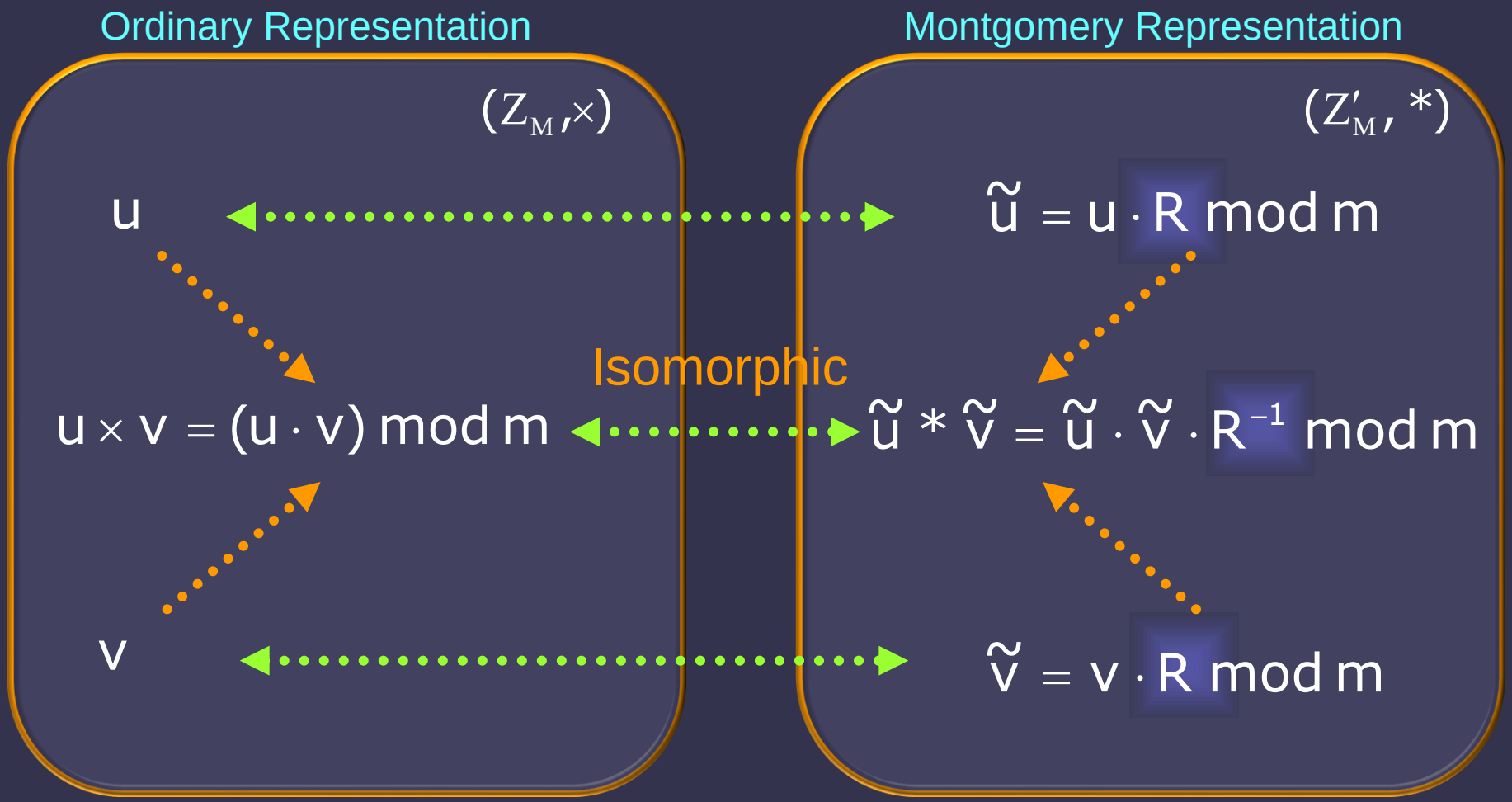
General overview



Sequential multiplications performed in Montgomery representation

Montgomery Multiplication

Montgomery radix $R = B^s > m, \ \gcd(R, m) = 1$



Montgomery Multiplication

Definition:

$$m : \text{large odd integer}$$

$$\tilde{u}, \tilde{v} \in \mathbf{Z}/m\mathbf{Z}\ , \gcd(m, B) = 1$$

$$\tilde{u} * \tilde{v} \triangleq \tilde{u} \cdot \tilde{v} \cdot R^{-1} \mod m$$

$$R > m \quad (\text{usually } R = B^s)$$

Sequential Computation on CPU

$$\tilde{u} * \tilde{v} = \tilde{u} \cdot \tilde{v} \cdot R^{-1} \mod m$$

Algorithm

$\tilde{z} = 0;$

for $(i = 0; i \le s - 1; i++)$

{

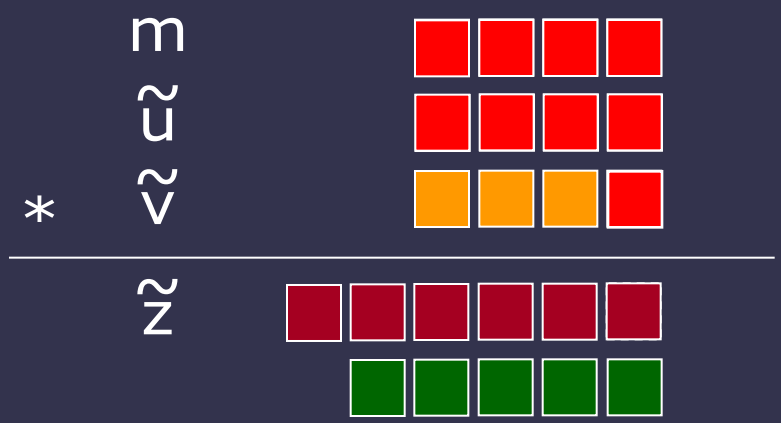
$\tilde{z} = \tilde{z} + \tilde{u} \cdot \tilde{v}_i;$

$q_M = (-\tilde{z}_0 \cdot m_0^{-1}) \bmod B;$

$\tilde{z} = (\tilde{z} + q_M \cdot m) \operatorname{div} B;$

}

if $\tilde{z} \ge m$ then $\tilde{z} = \tilde{z} - m;$



$$(\tilde{z} + (-\tilde{z}_0 \cdot m_0{}^{-1} \bmod B) \cdot m) \bmod B =$$

$$= (\tilde{z} + (-\tilde{z}_0 \cdot \cancel{m_0^{-1}} \cdot \cancel{m} \bmod B)) \bmod B$$

$$= (\tilde{z} - \tilde{z}_0) \bmod B = 0$$

Sequential Computation on CPU

$$\tilde{u} * \tilde{v} = \tilde{u} \cdot \tilde{v} \cdot R^{-1} \mod m$$

Algorithm

$\tilde{z} = 0;$

for $(i = 0; i \le s - 1; i++)$

{

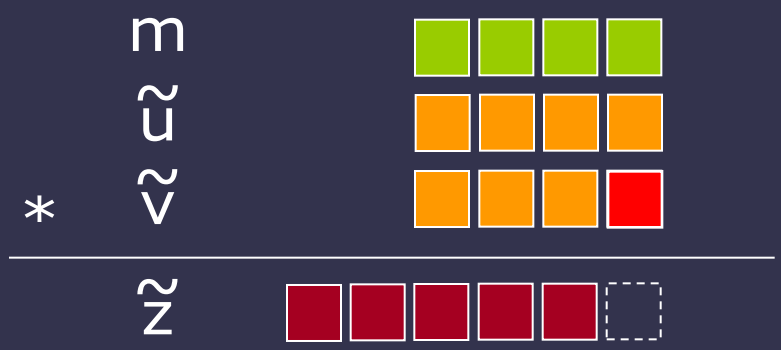
$\tilde{z} = \tilde{z} + \tilde{u} \cdot \tilde{v}_i;$

$q_M = (-\tilde{z}_0 \cdot m_0^{-1}) \mod B;$

$\tilde{z} = (\tilde{z} + q_M \cdot m) \text{ div } B;$

}

if $\tilde{z} \ge m$ then $\tilde{z} = \tilde{z} - m;$



$i = 0$

$$\tilde{z} < m \cdot (B - 1)$$

$$\tilde{z} < \frac{m \cdot (B - 1) + m \cdot (B - 1)}{B}$$

$$= \frac{2 \cdot m \cdot (B - 1)}{B} < 2m$$

Sequential Computation on CPU

$$\tilde{u} * \tilde{v} = \tilde{u} \cdot \tilde{v} \cdot R^{-1} \mod m$$

Algorithm

$\tilde{z} = 0;$

for $(i = 0; i \leq s - 1; i++)$

{

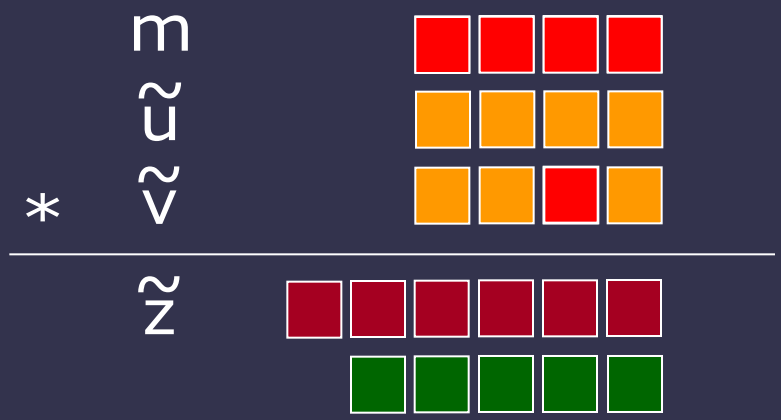
$\tilde{z} = \tilde{z} + \tilde{u} \cdot \tilde{v}_i;$

$q_M = (-\tilde{z}_0 \cdot m_0^{-1}) \bmod B;$

$\tilde{z} = (\tilde{z} + q_M \cdot m) \operatorname{div} B;$

}

if $\tilde{z} \geq m$ then $\tilde{z} = \tilde{z} - m;$



Sequential Computation on CPU

$$\tilde{u} * \tilde{v} = \tilde{u} \cdot \tilde{v} \cdot R^{-1} \mod m$$

Algorithm

$\tilde{z} = 0;$

for $(i = 0; i \le s - 1; i++)$

{

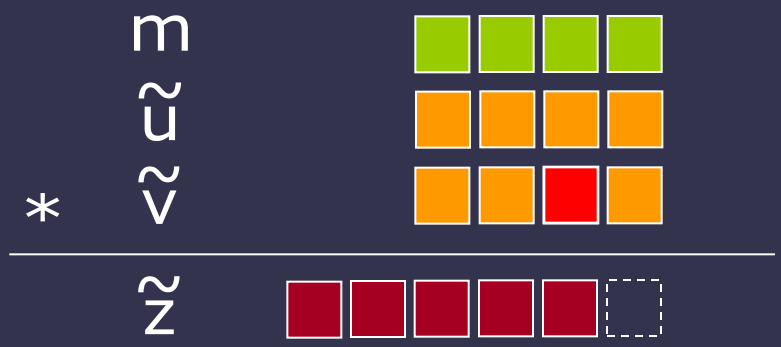
$\tilde{z} = \tilde{z} + \tilde{u} \cdot \tilde{v}_i;$

$q_M = (-\tilde{z}_0 \cdot m_0^{-1}) \mod B;$

$\tilde{z} = (\tilde{z} + q_M \cdot m) \operatorname{div} B;$

}

if $\tilde{z} \ge m$ then $\tilde{z} = \tilde{z} - m;$



$i = 1$

$$0 \le \tilde{z} < 2 \cdot m$$

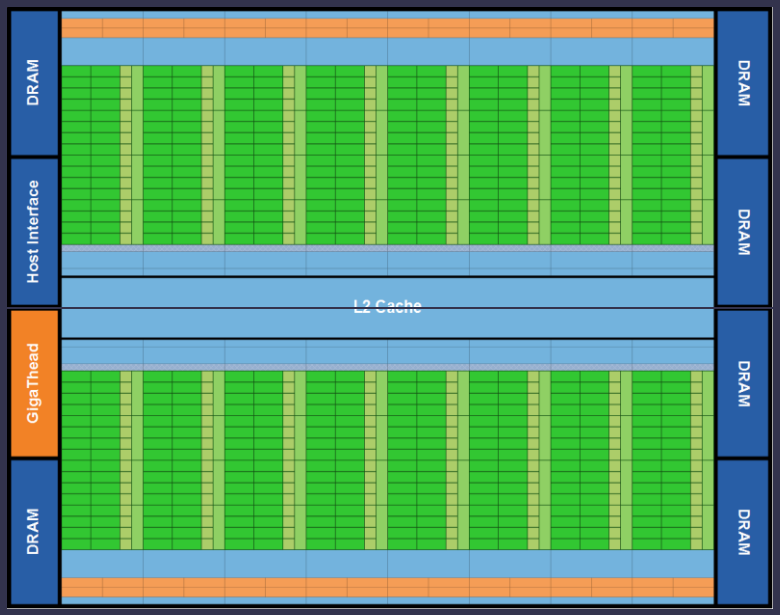
$$\tilde{z} < 2 \cdot m + m \cdot (B - 1)$$

$$\tilde{z} < \frac{2 \cdot m + m \cdot (B - 1) + m \cdot (B - 1)}{B}$$

$$= \frac{m \cdot (2 + B - 1 + B - 1)}{B} < 2 \cdot m$$

Fermi architecture

- Specifications:
  - 3 billon transistors
  - 16 Streaming Multiprocessors (SM)
  - 6 x 64-bit memory partitions
    Up to total 6GB GDDR5 with ECC
  - GigaThread global scheduler
  - Shared L2 Cache (768KB)

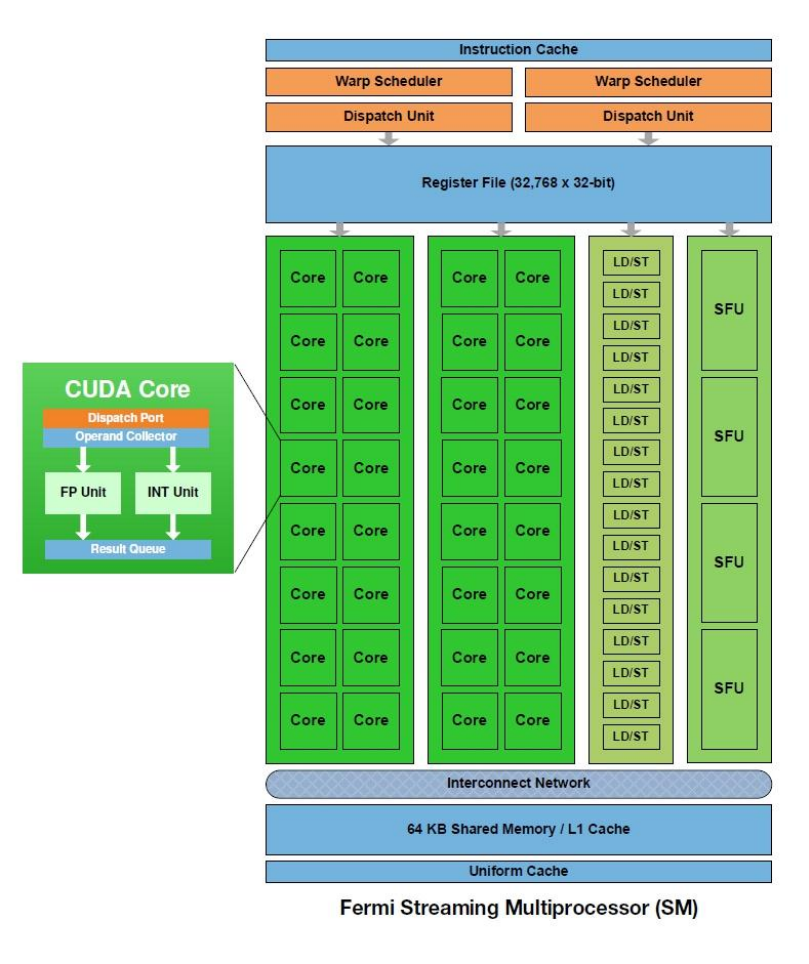


Source: NVIDIA's next Generation CUDA™ Compute Architecture: Fermi

Fermi architecture

- Streaming Multiprocessor
  - 32 CUDA Cores (16 x 32 = 512)
  - Dual warp scheduler
  - 16 LD/ST Units
  - 4 Special Function Units (SFU)
  - 64KB of configurable Shared Memory and L1 Cache (48KB/16KB)

- CUDA Core
  - Pipelined ALU and FPU
  - ALU supports 32-bit int
  - FPU single precision (512 FMA ops / clock)
  - 1K 32-bit registers per core

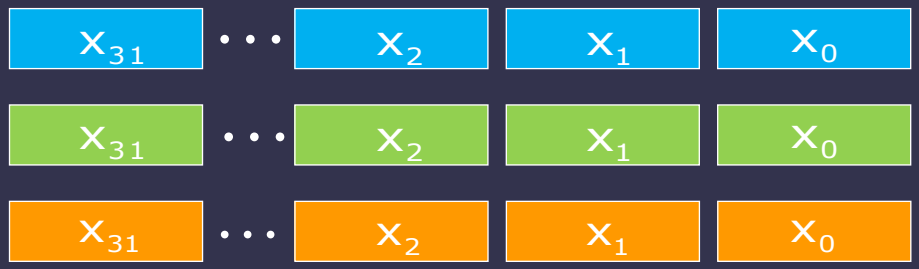


Fermi Streaming Multiprocessor (SM)

Source: NVIDIA's next Generation CUDA™ Compute Architecture: Fermi

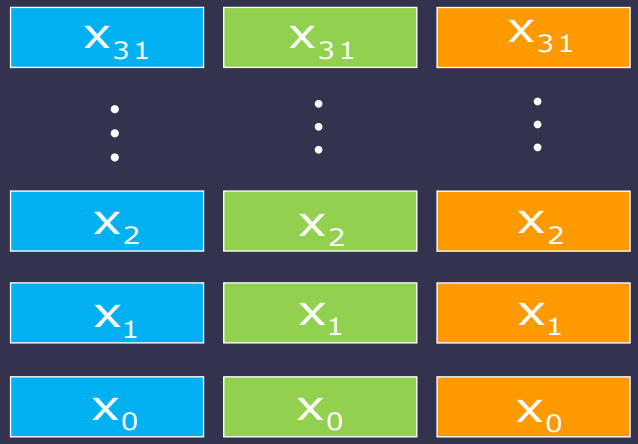
Representation of Integers

Parallel version



- Low Latency
- Cryptography

Sequential version



- High Latency
- Cryptanalysis

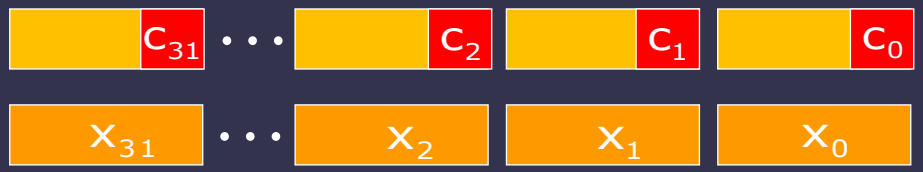
Representation of Integers

- To avoid barriers (mem fence) try to fit entire operand within a block of 32 threads (Warps)

  Data coherence is maintained within a warp.

- Each thread operates in one limb in radix $B=2^{32}$

- Possible representations:
  - Avizienis representation (signed-digit)
  - Residue Number System
  - Carry-save

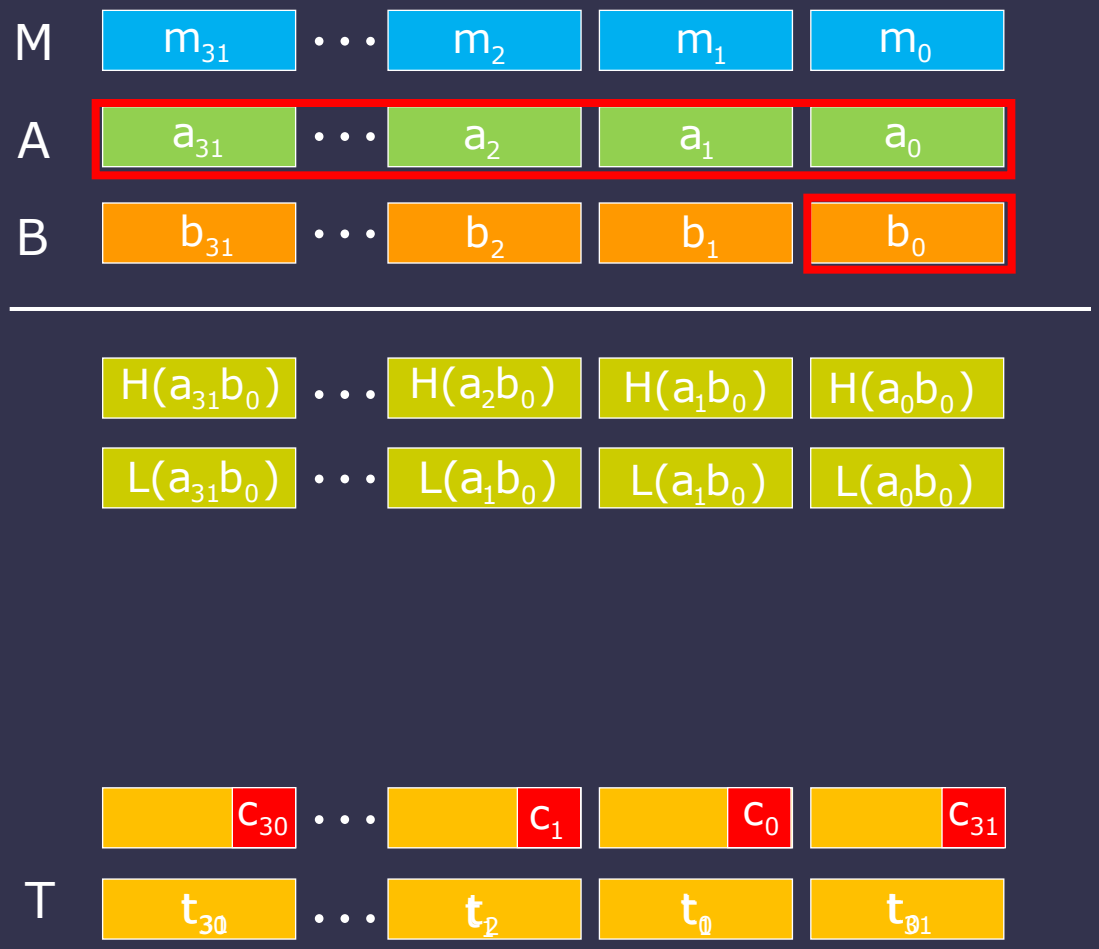


Montgomery Multiplication

$$\tilde{u} * \tilde{v} = \tilde{u} \cdot \tilde{v} \cdot R^{-1} \mod m$$

Algorithm

$A := \tilde{u}; B := \tilde{v}; M := m;$
$T := 0;$
for $(i = 0; i \le s - 1; i++)$
{
  $T := T + b_i A;$
  $q_M := t_0 \cdot (-m_0)^{-1} \mod B;$
  $T := (T + q_M \cdot M) \text{ div } B;$
}
if $T \ge M$ then $Z := T - M;$
else $Z := T;$

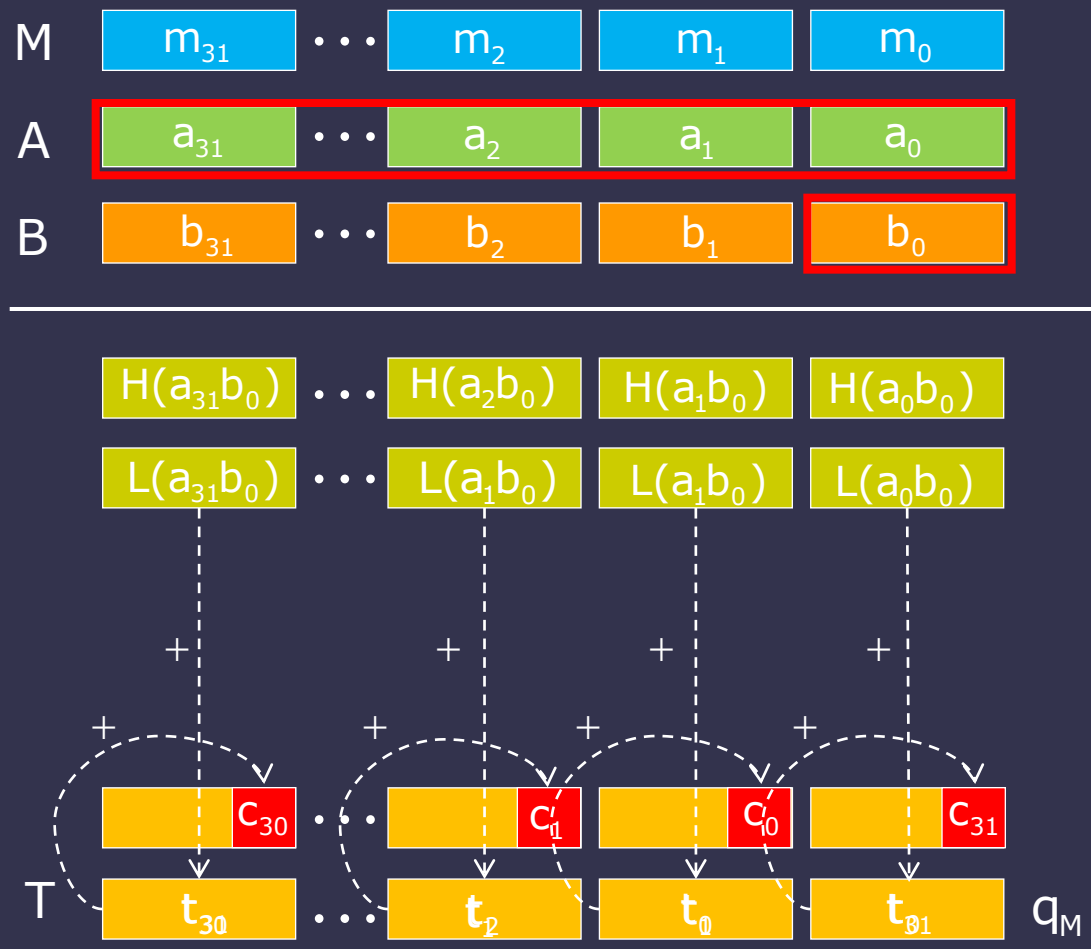


Montgomery Multiplication

$$\tilde{u} * \tilde{v} = \tilde{u} \cdot \tilde{v} \cdot R^{-1} \mod m$$

Algorithm

$A := \tilde{u}; B := \tilde{v}; M := m;$
$T := 0;$
for $(i = 0; i \le s - 1; i++)$
{
  $T := T + b_i A;$
  $q_M := t_0 \cdot (-m_0)^{-1} \mod B;$
  $T := (T + q_M \cdot M) \text{ div } B;$
}
if $T \ge M$ then $Z := T - M;$
else $Z := T;$

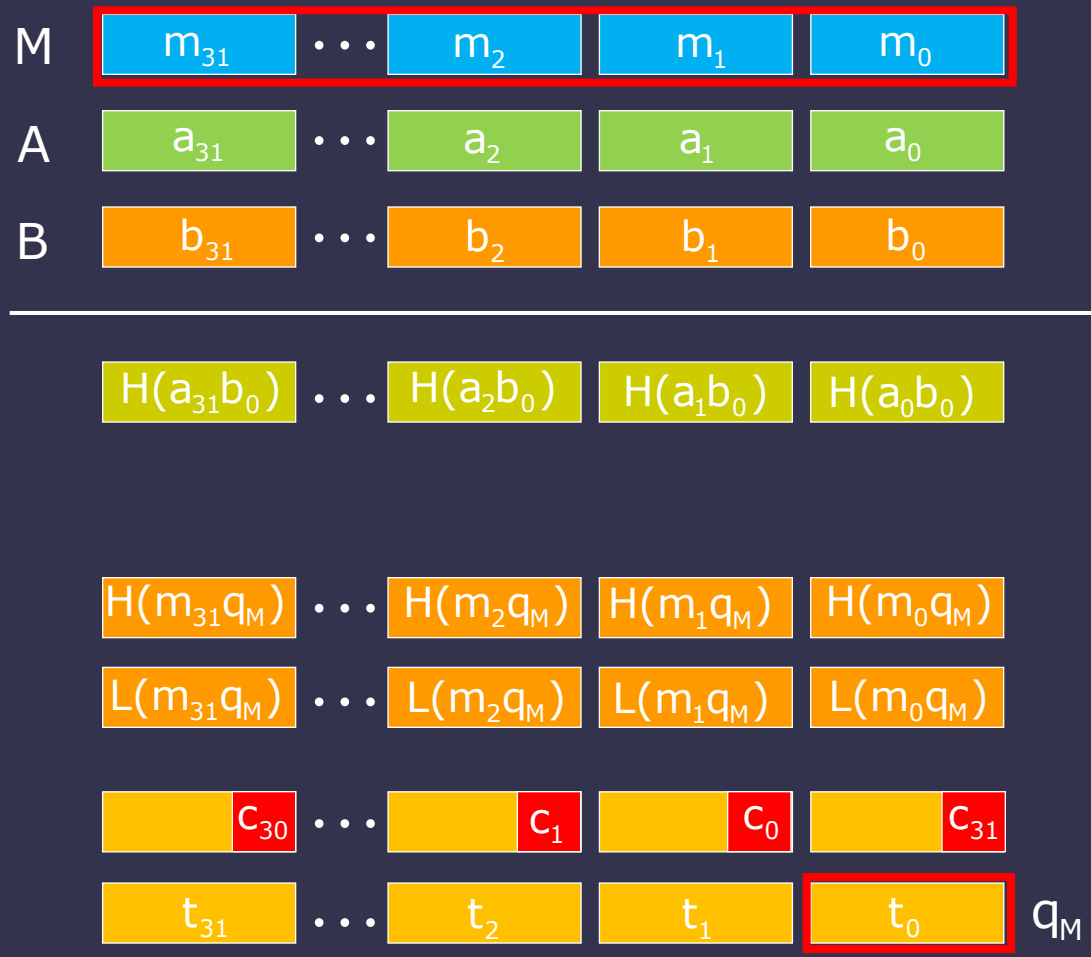


Montgomery Multiplication

$$\tilde{u} * \tilde{v} = \tilde{u} \cdot \tilde{v} \cdot R^{-1} \mod m$$

Algorithm

$A := \tilde{u}; B := \tilde{v}; M := m;$
$T := 0;$
$\text{for } (i = 0; i \le s - 1; i++)$
$\{$
$\quad T := T + b_i A;$
$\quad q_M := t_0 \cdot (-m_0)^{-1} \mod B;$
$\quad T := (T + q_M \cdot M) \text{ div } B;$
$\}$
$\text{if } T \ge M \text{ then } Z := T - M;$
$\text{else } Z := T;$

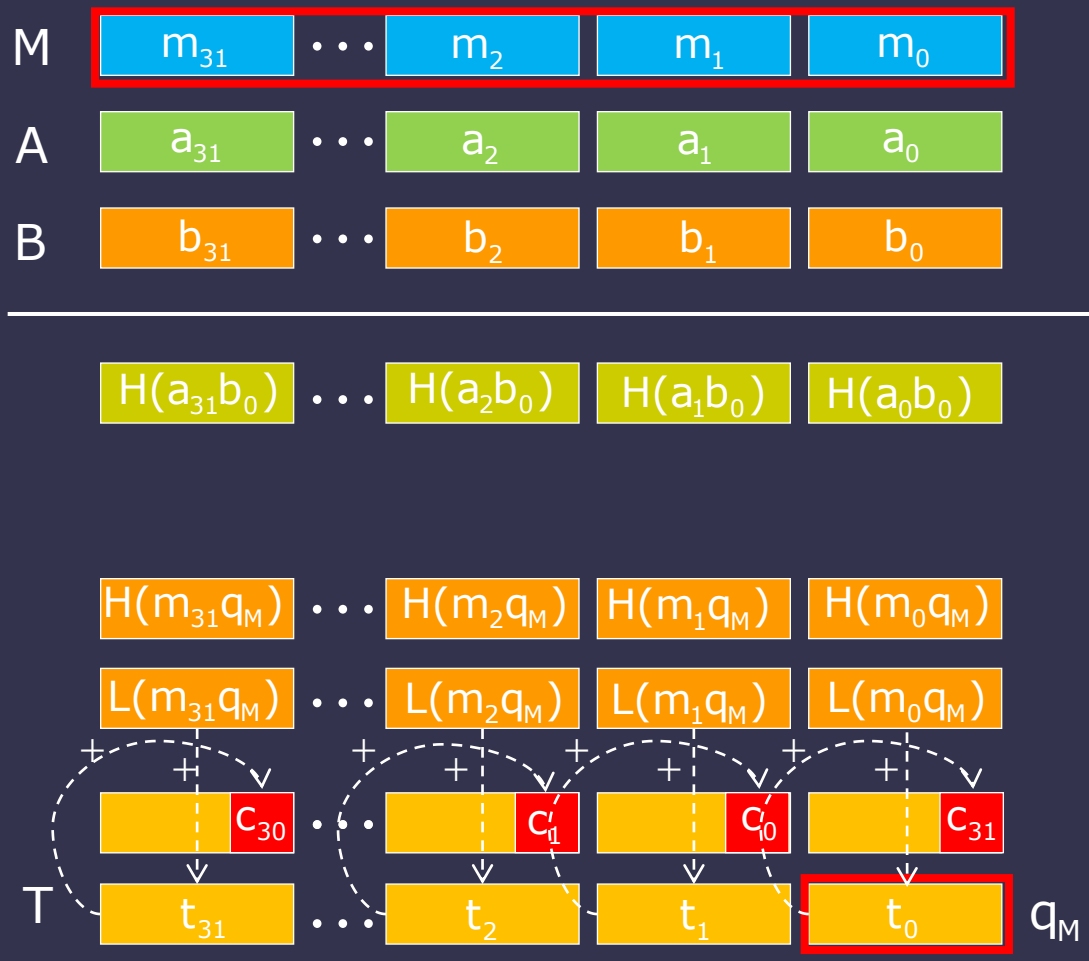


Montgomery Multiplication

$$\tilde{u} * \tilde{v} = \tilde{u} \cdot \tilde{v} \cdot R^{-1} \mod m$$

Algorithm

$A := \tilde{u}; B := \tilde{v}; M := m;$
$T := 0;$
$for\ (i = 0; i \le s - 1; i++)$
{
  $T := T + b_i A;$
  $q_M := t_0 \cdot (-m_0)^{-1} \mod B;$
  $T := (T + q_M \cdot M)\ div\ B;$
}
$if\ T \ge M\ then\ Z := T - M;$
$else\ Z := T;$

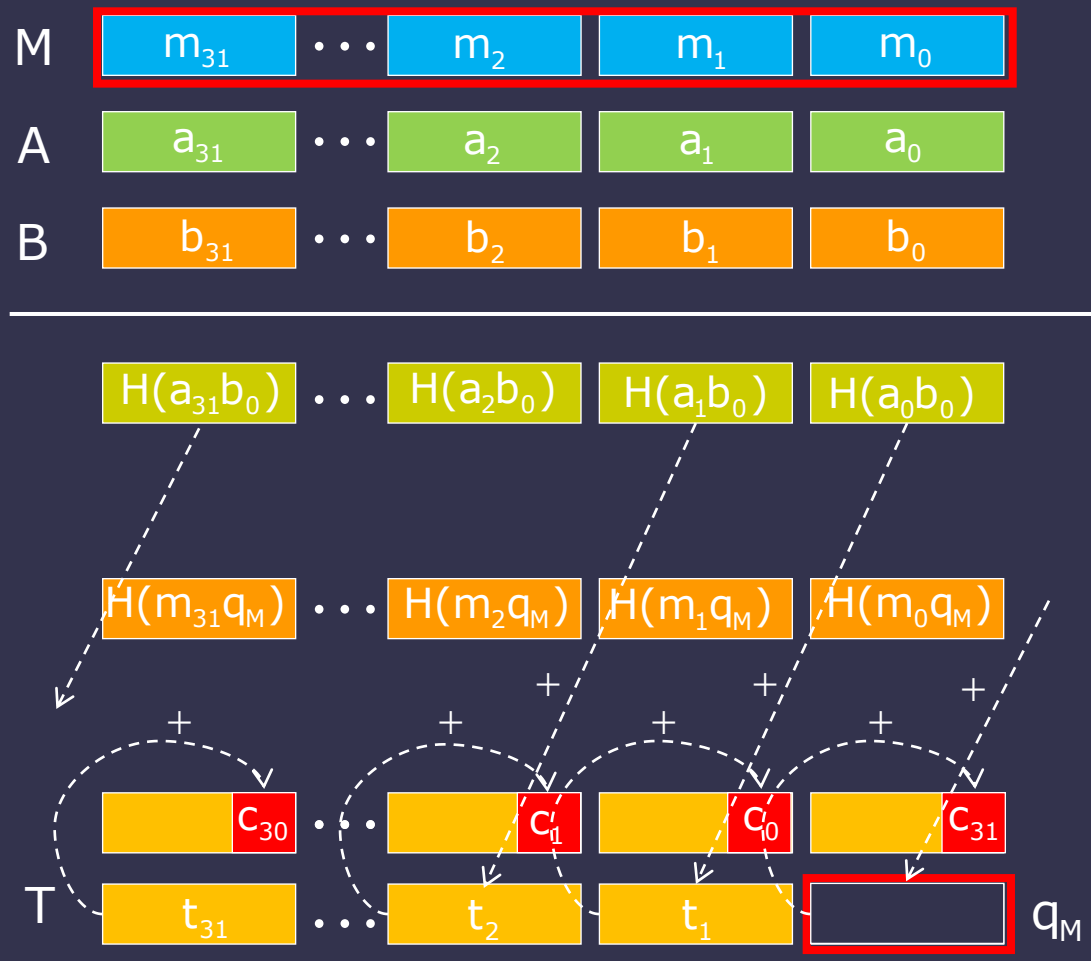


Montgomery Multiplication

$$\tilde{u} * \tilde{v} = \tilde{u} \cdot \tilde{v} \cdot R^{-1} \mod m$$

Algorithm

```
A := ũ; B := ṽ; M := m;
T := 0;
for (i = 0; i ≤ s - 1; i++)
{
  T := T + b_i A;
  q_M := t_0 · (−m_0)^-1 mod B;
  T := (T + q_M · M) div B;
}
if T ≥ M then Z := T - M;
else Z := T;
```

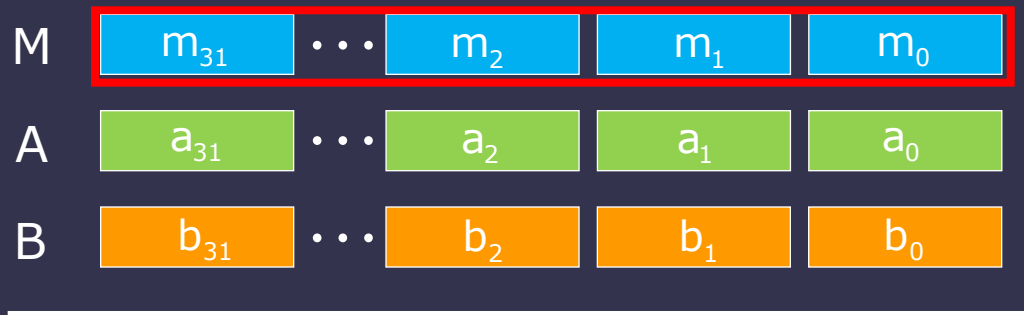


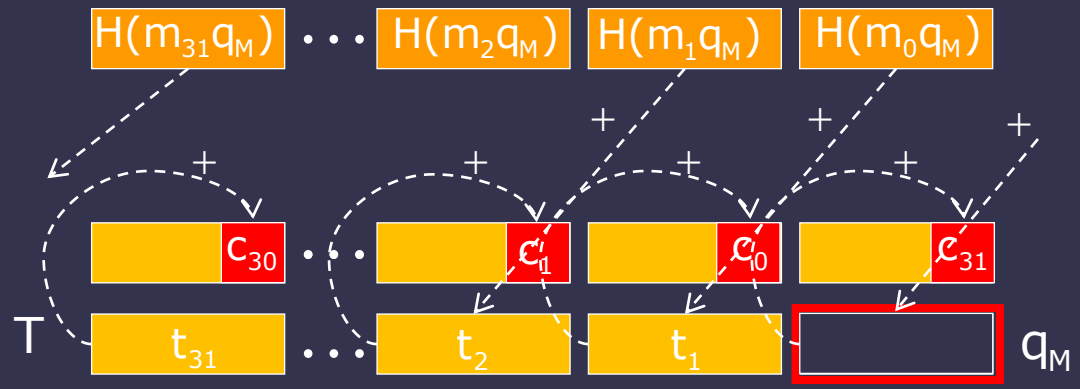
Montgomery Multiplication

$$\tilde{u} * \tilde{v} = \tilde{u} \cdot \tilde{v} \cdot R^{-1} \mod m$$

Algorithm

```
A := ũ; B := ṽ; M := m;
T := 0;
for (i = 0; i ≤ s - 1; i++)
{
  T := T + b_i A;
  q_M := t_0 · (−m_0)^-1 mod B;
  T := (T + q_M · M) div B;
}
if T ≥ M then Z := T - M;
else Z := T;
```



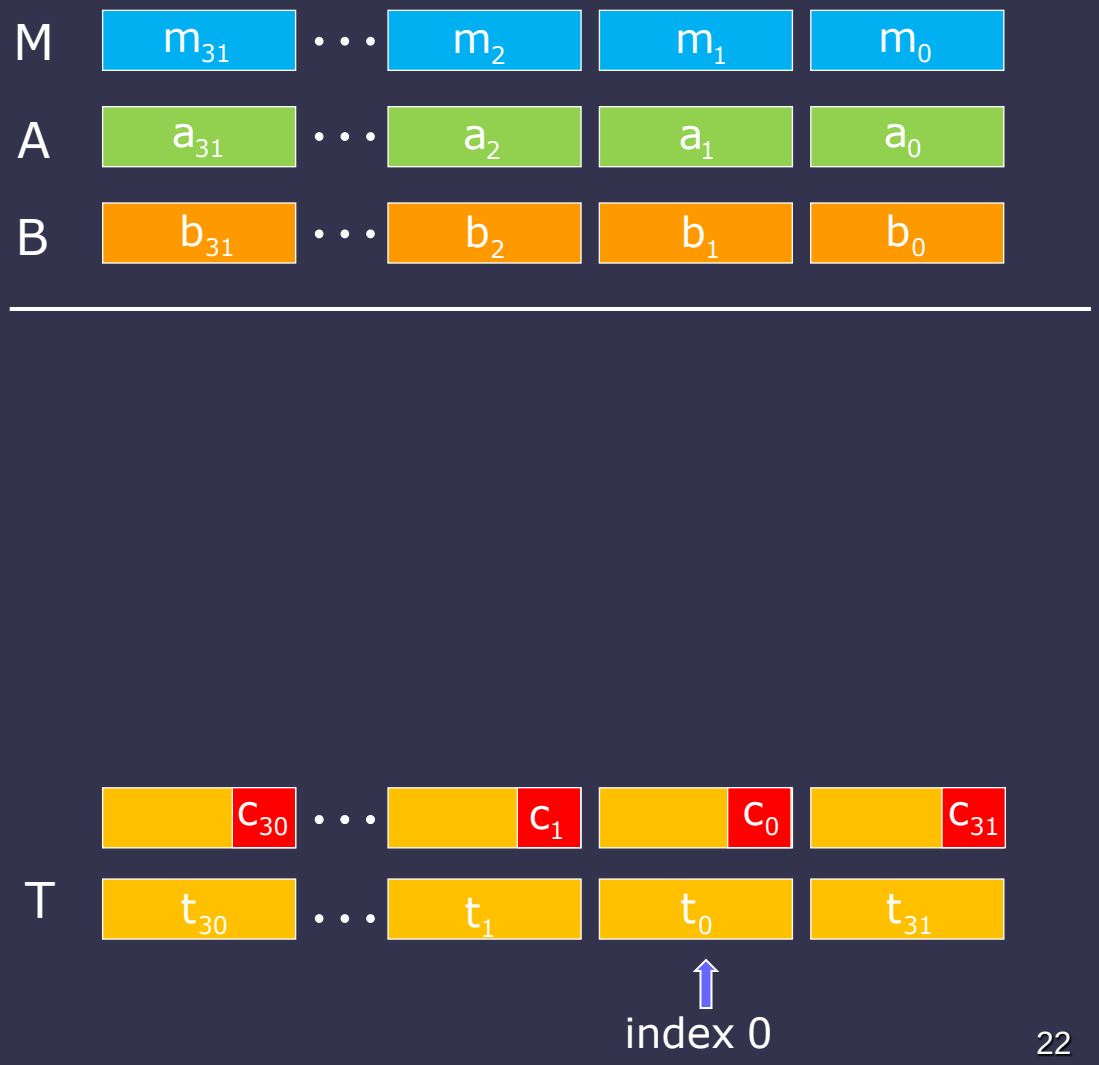


Montgomery Multiplication

$$\tilde{u} * \tilde{v} = \tilde{u} \cdot \tilde{v} \cdot R^{-1} \mod m$$

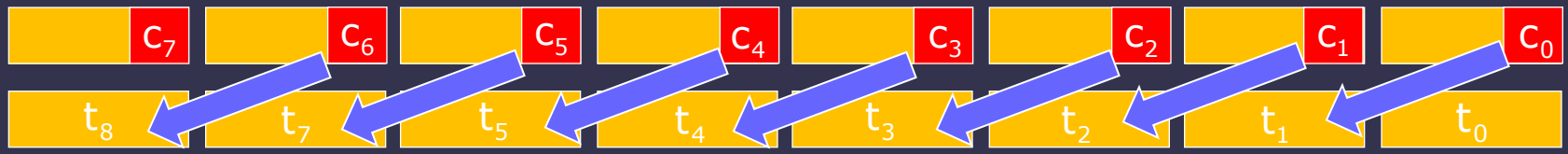
Algorithm

```
A := ũ; B := ṽ; M := m;
T := 0;
for (i = 0; i ≤ s - 1; i++)
{
  T := T + b_i A;
  q_M := t_0 · (−m_0)^-1 mod B;
  T := (T + q_M · M) div B;
}
if T ≥ M then Z := T - M;
else Z := T;
```



Carry propagation

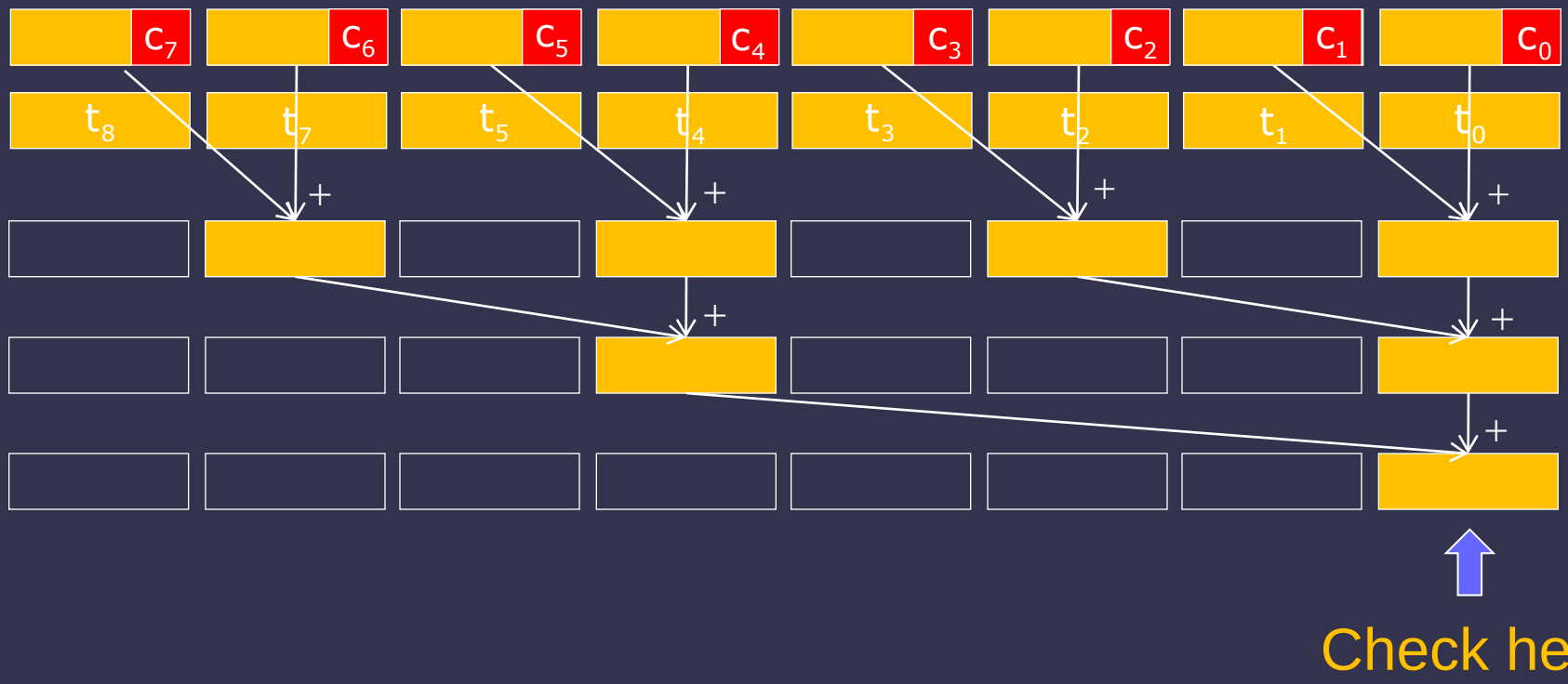
- Requires 31 iteration of additions with carry



- Probability of new carry after one iteration is very low.
- One time carry propagation + logarithmic time verification.

Avoiding carry propagation

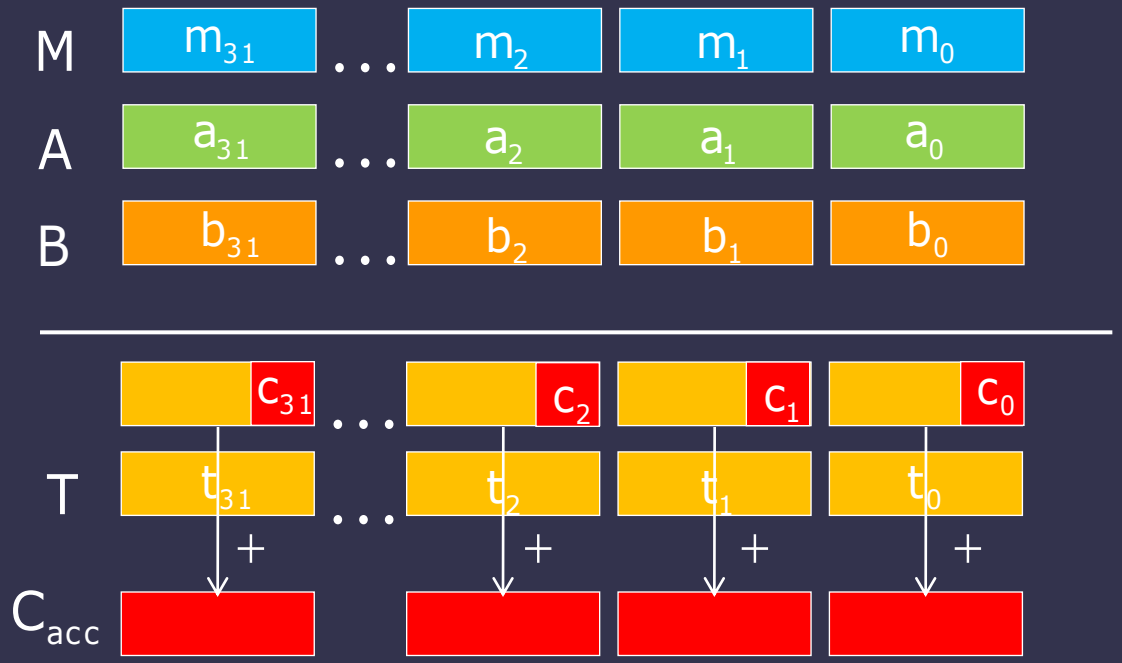
- Remaining carries can be checked in log time



- Probability of new carry after one iteration is very low.
- One time carry propagation + logarithmic time verification.
- If carry detected, continue carry propagation

Speeding up further

- Prepare two exponentiation algorithms:
  - 1) *Fast:* Accumulate carries, check after exponentiation
  - 2) *Slow:* Checks every modular multiplication



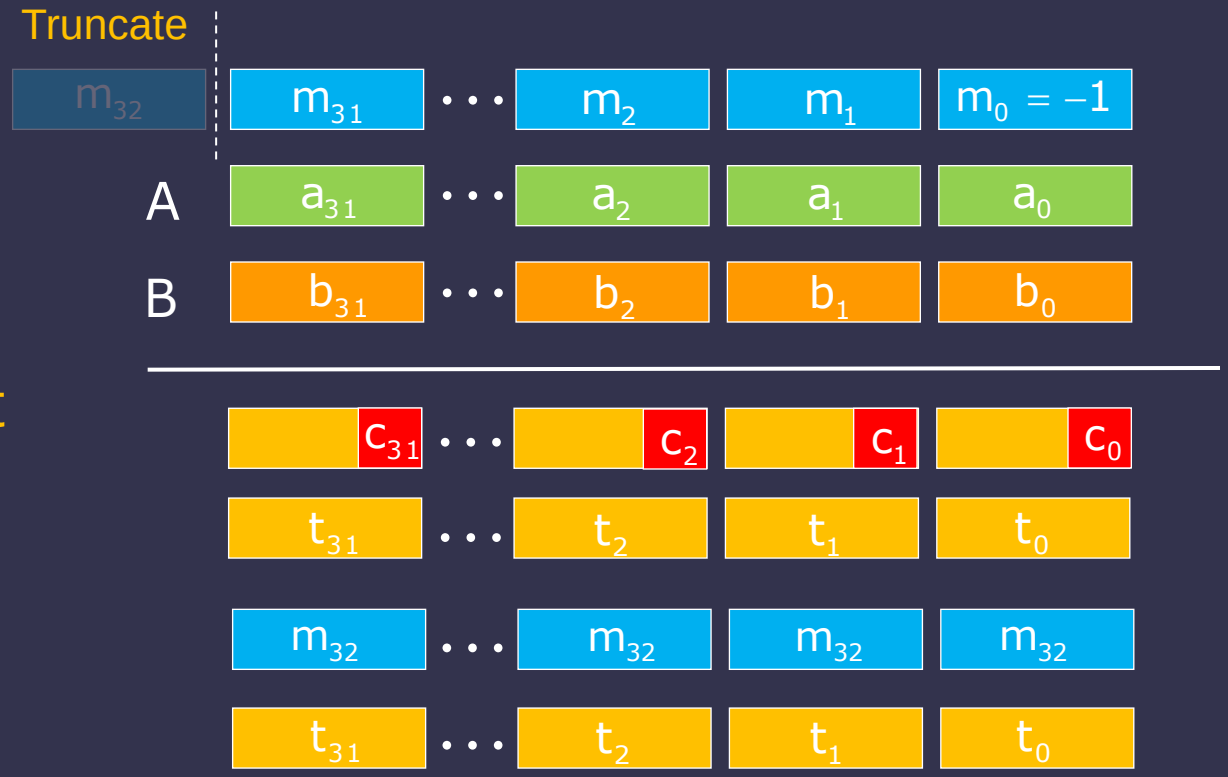
- Use log time check after exponentiation
- If carry detected in the end, call slow exponentiation

Operand Scaling Techniques

- *Scaling the modulus*

$\tilde{M} = M \cdot ((-m_0)^{-1} \bmod B)$

$q_M := t_0;$



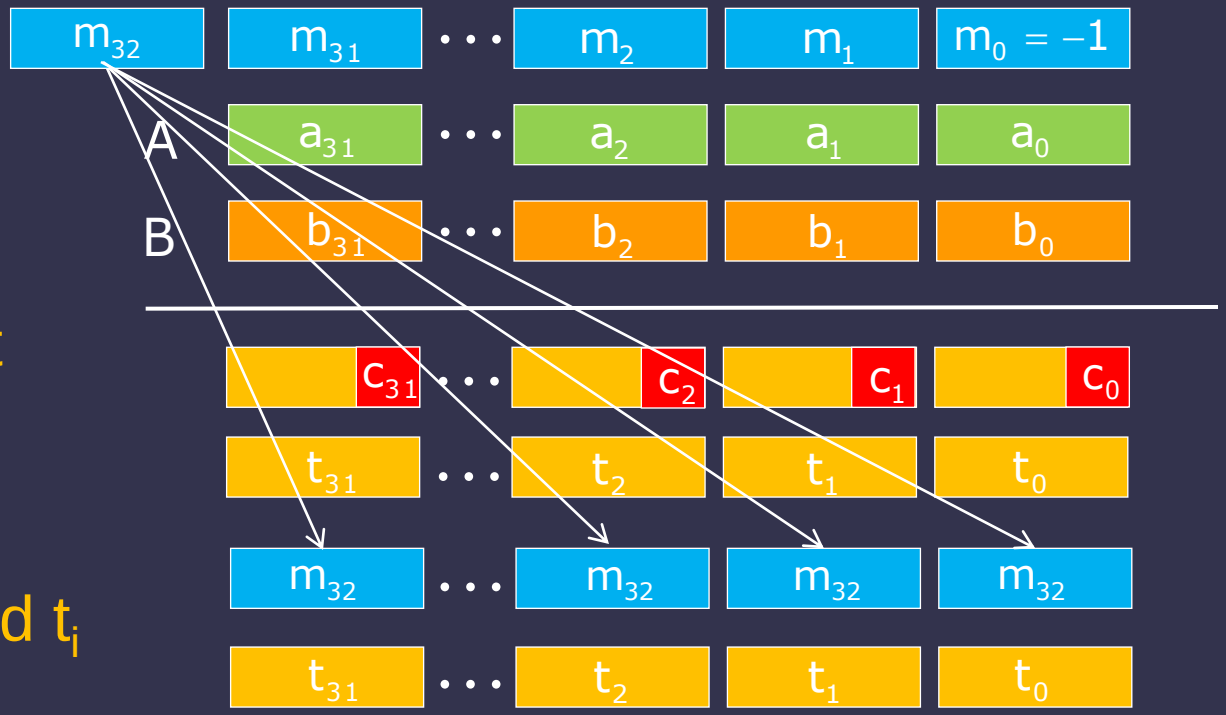
- Simplifies quotient determination

Operand Scaling Techniques

- *Scaling the modulus*

$$\tilde{M} = M \cdot ((-m_0)^{-1} \bmod B)$$

$$q_M := t_0;$$

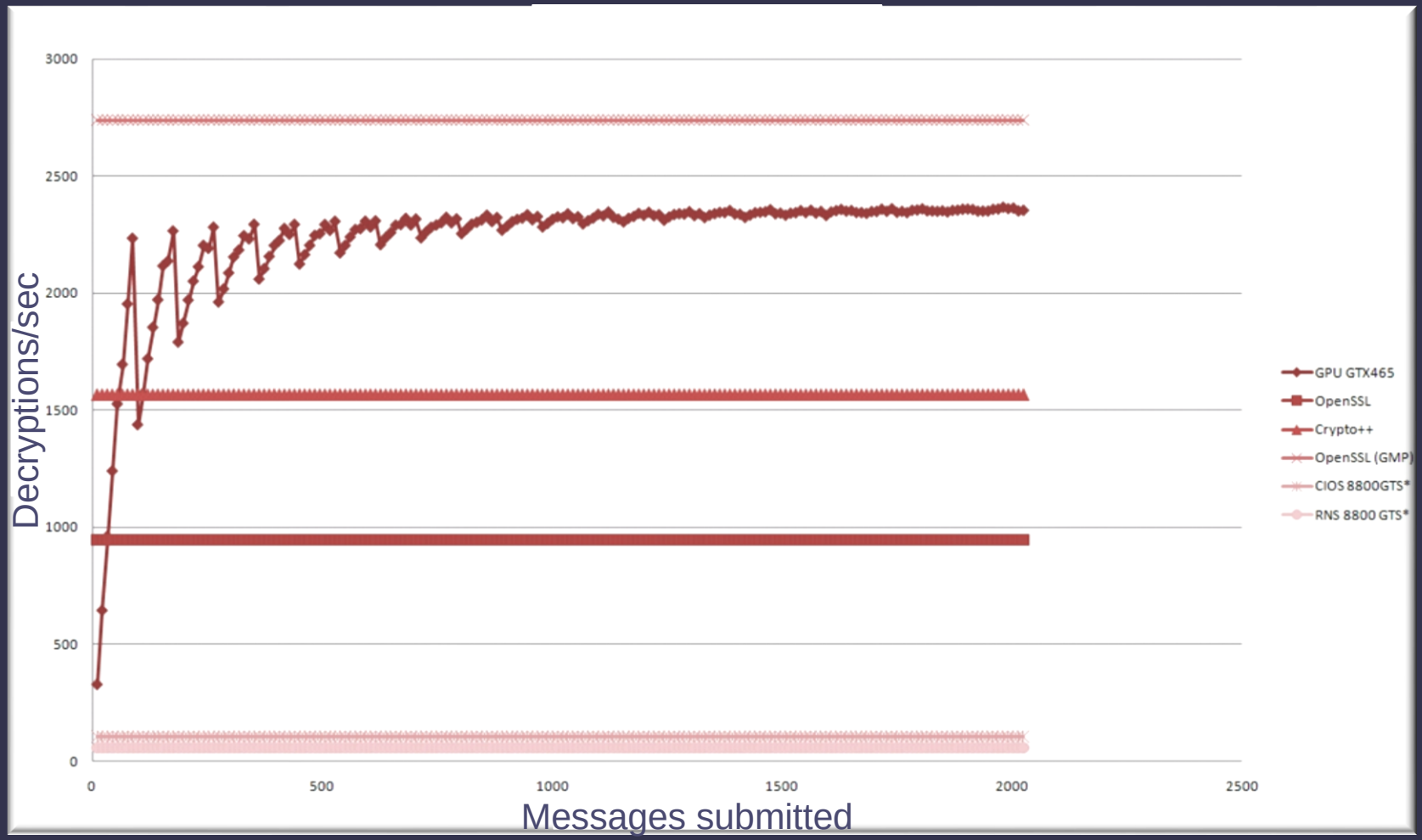


- Simplifies quotient determination
- *Drawback*
  - Needs to record t_i
  - Result < 3M
- Similar case if multiplicand A is scaled up by radix B

Performance Evaluation

- *Evaluation Platform*
  - Linux x86_64
  - OpenCL 1.0 CUDA
  - Nvidia GeForce GTX 465
  - 11 SM with 32 cores each = 352 Cores
    (5 SM are disabled to increase yield production)
  - Device clock freq. 1.2 GHz
    (lower clock freq. compared to GTX480@1.4GHz)
  - Measurements includes I/O
    (CRT is performed in GPU)

Performance Evaluation



AMD Opteron™ 1381 Quad-Core @ 2.6GHz

* R. Szerwinski and T. Guneysu, “Exploiting the Power of GPUs for Asymmetric Cryptography”, CHES 2008

Performance Evaluation

	OpenSSL Normal[1]	Crypto++[1]	OpenSSL GMP [1]	GPU 8800 GTS (CIOS) [3]	GPU 8800 GTS (RNS) [3]	GPU GTX465 [2]
Ops/sec	946.9	1'566.28 (scaled)	2'738.9	104.3	57.9	2'232.43
Delay [ms]	-	-	-	55'184	849	39.4

(1) Evaluated on AMD Opteron™ 1381 Quad-Core @ 2.6GHz on Linux x86_64

(2) Current implementation on Nvidia GTX465 (11 MS, total of 352 CUDA Cores) @ 1.2GHz

(3) R. Szerwinski and T. Guneysu, "Exploiting the Power of GPUs for Asymmetric Cryptography", CHES 2008

Implemented on Nvidia 8800 GTS (Total 112 CUDA Cores @ 1.5GHz)

Further optimizations (in progress)

- *Carry generation*

Code:

```
b [ local_id ] +=  a[ local_id ];
c [ local_id ] += ( b [ local_id ] < a [ local_id ] );
```

  - Fermi architecture supports addition with carry (add.cc)
  - Inline assembly using CUDA

- *Exponentiation algorithm*

  - Currently using left-to-right binary exponentiation
  - With windows exponentiation 25% speed up
  - Randomization on checking point for carries as countermeasure

Summary

- I presented an implementation of RSA 2048 on GPUs that takes advantage of data coherence inside the warp.
- Current implementation is competitive compared to CPU implementations and suitable for server applications with low latency.
- The use of GPUs as cryptographic accelerators seems to have a promising future.