An Implementation of RSA 2048 on GPUs Using CUDA

Marcelo E. Kaihara
EPFL IC IIF LACAL
Feb. 8, 2011

RAIM'11 Perpignan

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

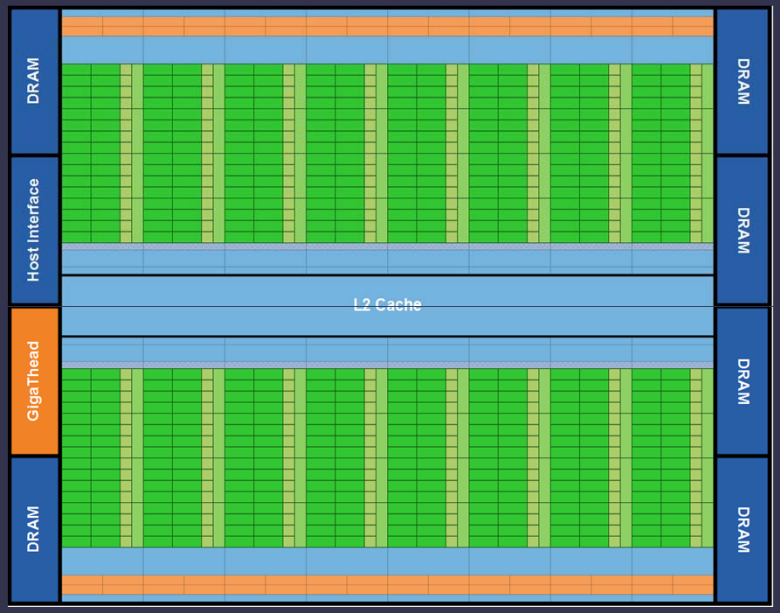
- Use GPUs as cryptographic accelerators to offload work from the CPU.
- Aim for server applications

- Low Latency
- High throughput

- Parallel algorithm
- Inline PTX with CUDA
  (near assembly)

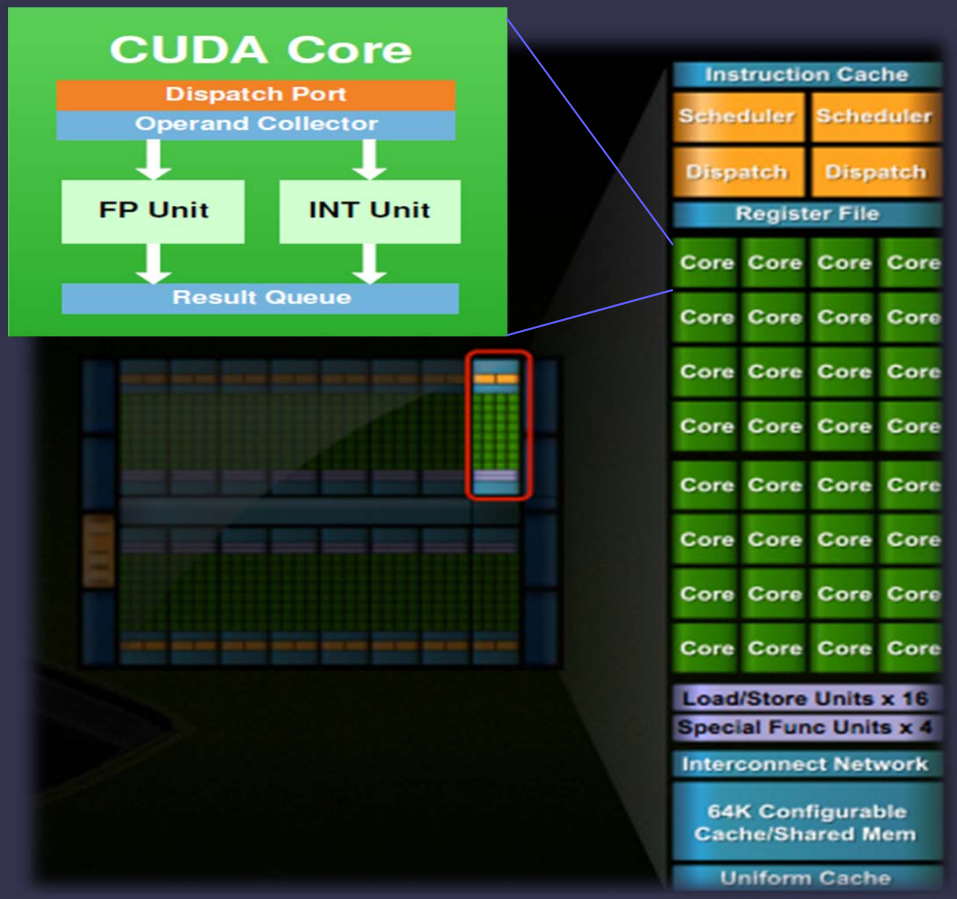
Fermi architecture

- 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



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

Streaming Multiprocessor

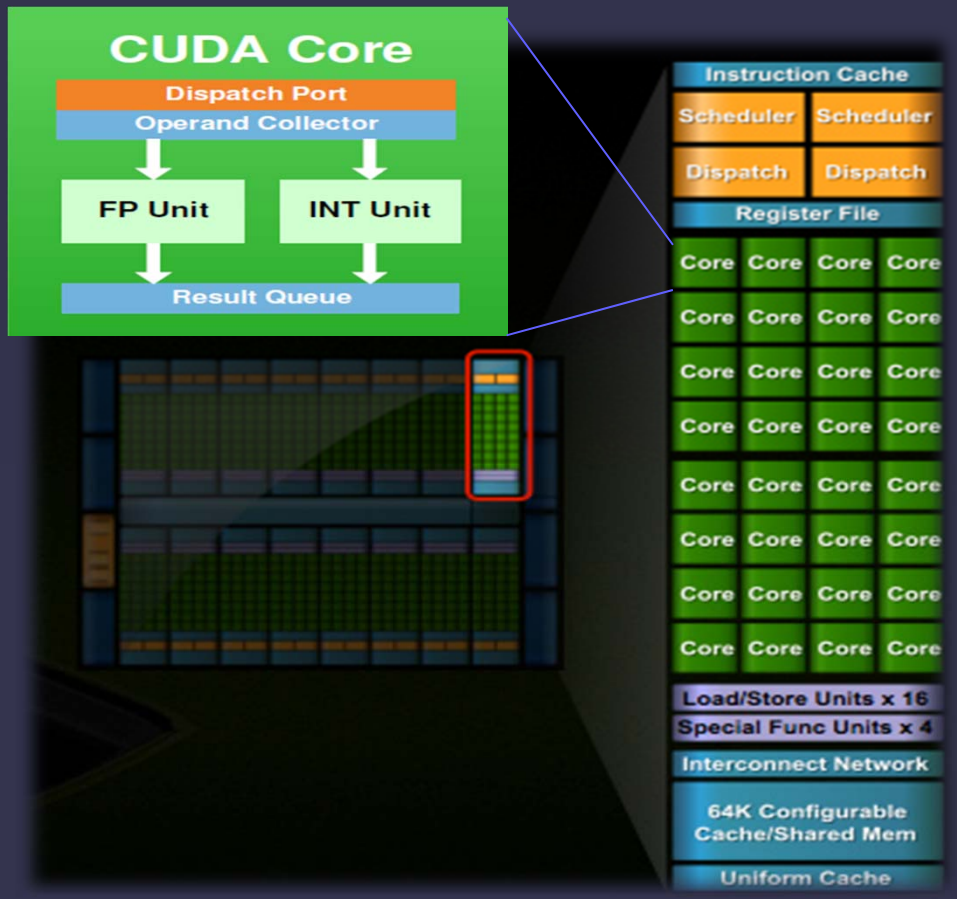
- 32 CUDA Cores (16 x 32 = 512)
- Dual warp scheduler
- 16 LD/ST Units
- 4 Special Function Units (SFU)
- 64KB of configurable Shared Memory and L1 Cache (48KB / 16KB)

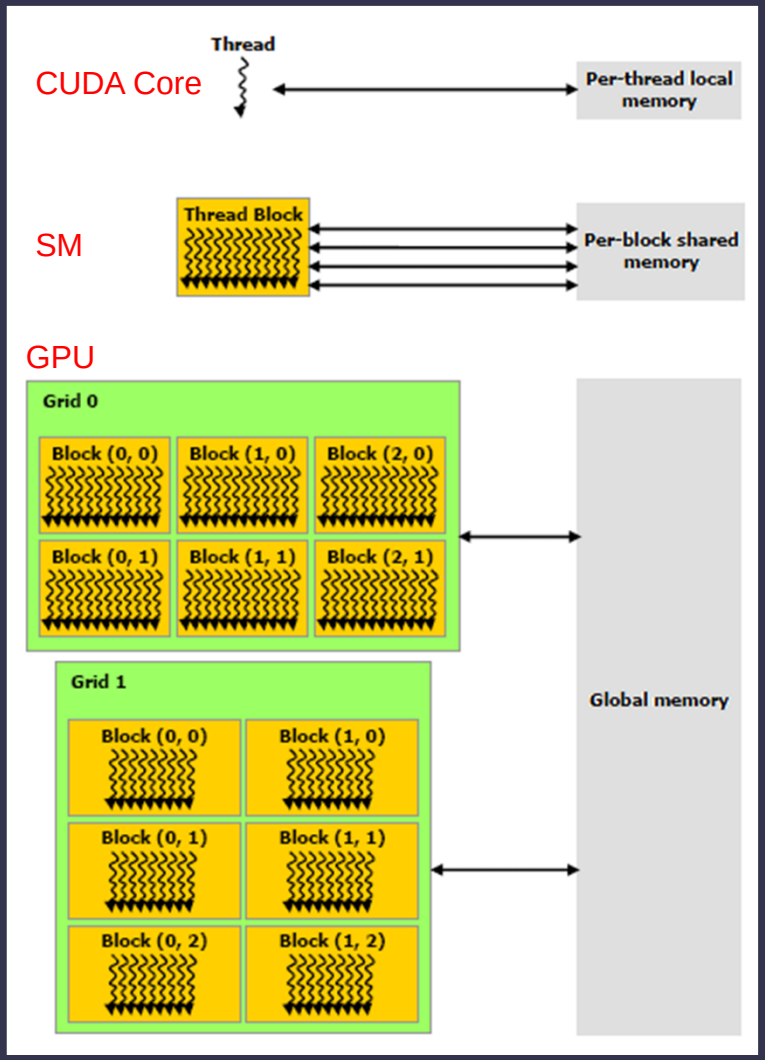
CUDA Core

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

CUDA architecture

Compute Unified Device Architecture





- SIMT = Single Instruction Multiple Thread
  (it is SIMD inside a Warp)
- Thread executed in a CUDA Core
- Thread Block executed in a Streaming Multiprocessor (SM)
- Grid of Thread Bocks executed on GPU

Source: CUDA Programming Guide Version 2.3.1

RSA 2048 Decryption

Decryption

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

Precomputed values

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

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

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

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

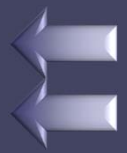
Chinese Remainder Theorem

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

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

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

$z = z_2 + h \cdot q$

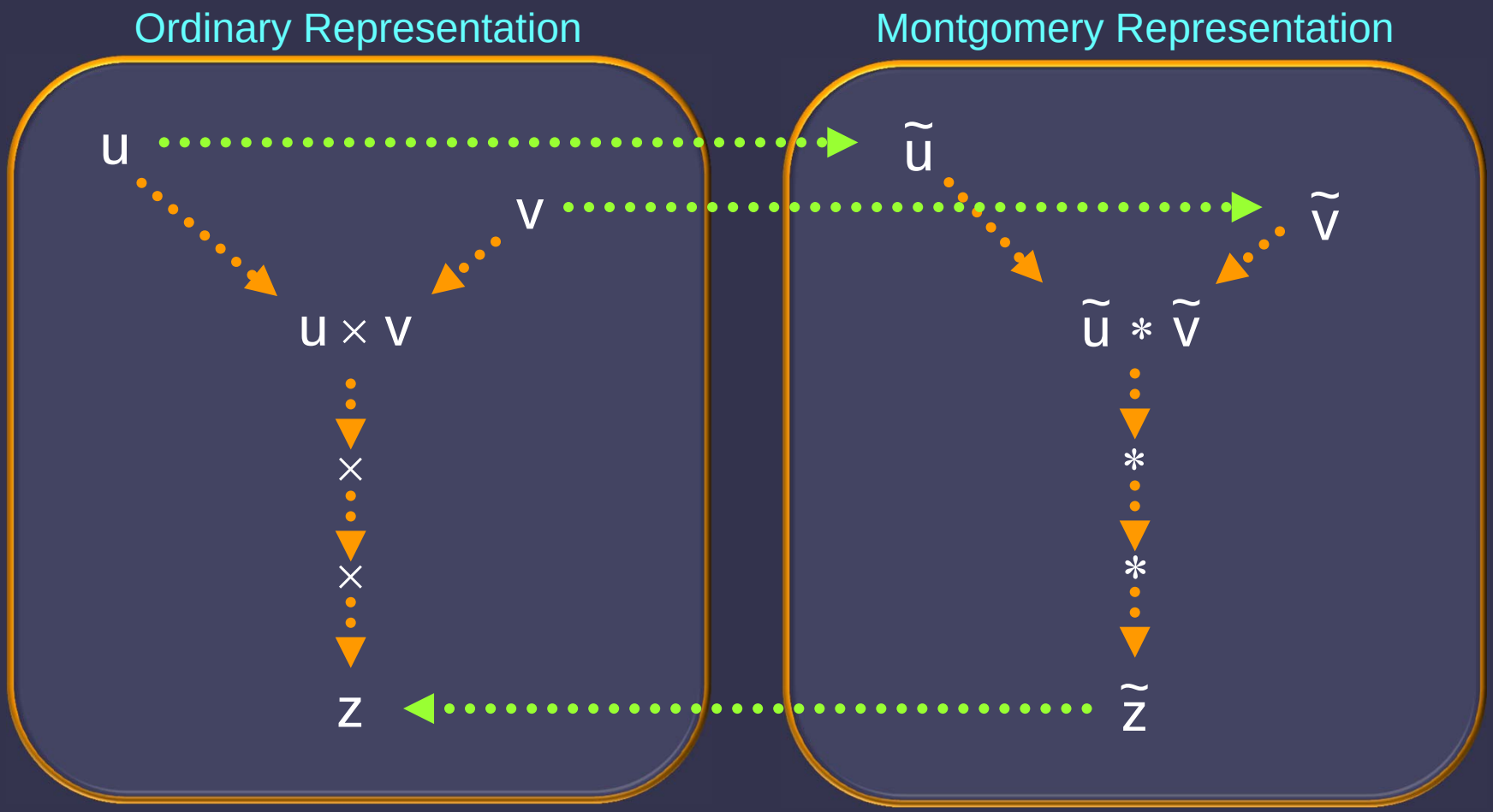


Mod Exp 1024 moduli
(32 limbs of 32-bits)

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

Montgomery Multiplication

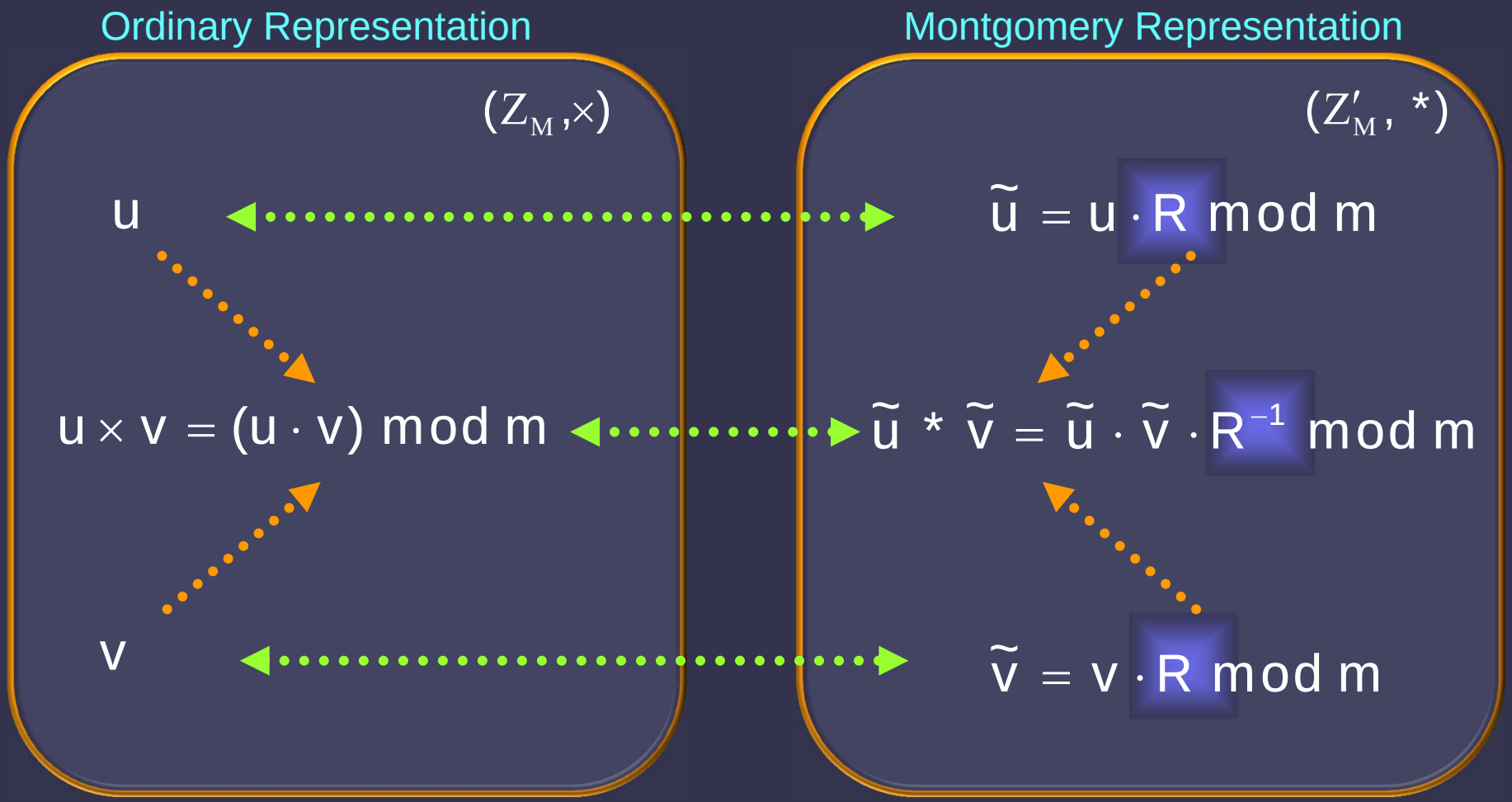
General overview



Sequential multiplications performed in Montgomery representation

Montgomery Multiplication

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



Montgomery Multiplication

Definition:

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

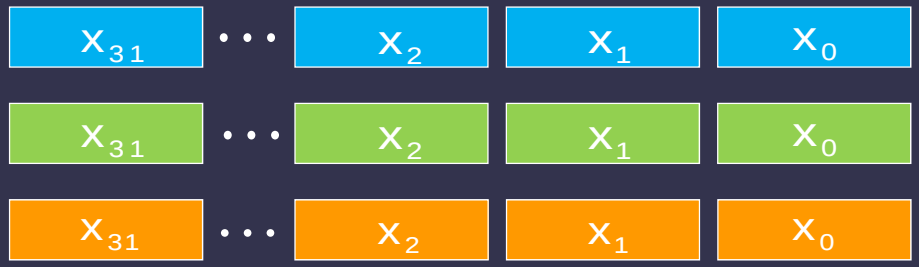
$$\tilde{u}, \tilde{v} \in \mathbf{Z}/m\mathbf{Z}\ , \gcd(m, B) = 1$$

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

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

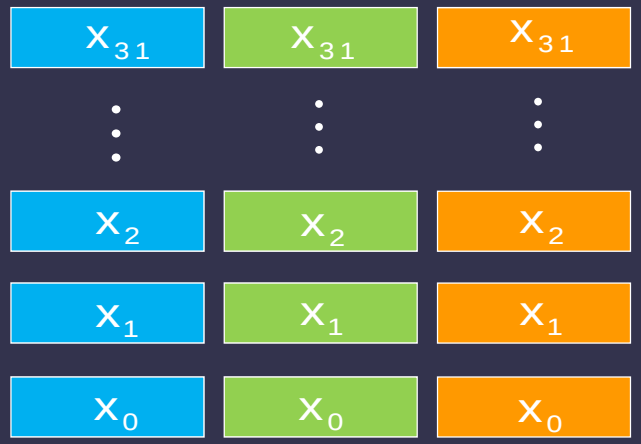
Representation of Integers

Parallel version



$s = 32$ $X = \sum_{i=0}^{s-1} x_i B^i$ $B = 2^{32}$

Serial version



- Low Latency
- Easy: Windows Exp
- Difficult: Karatsuba / Squaring
- Cryptography

- High Latency
- Easy: Karatsuba / Squaring
- Difficult: Windows Exp
- Cryptanalysis

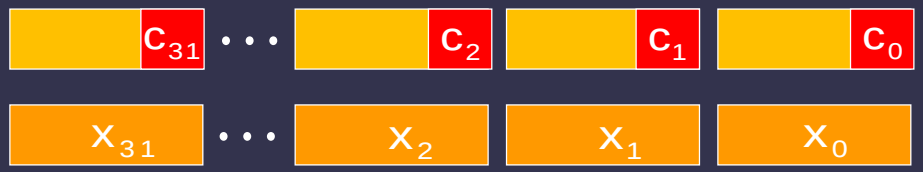
Representation of Integers

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

  Data coherence is maintained within a warp.

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

- Possible representations:
  - Signed-digit representation
  - Residue Number System
  - Carry-save (extra vector to store carries)

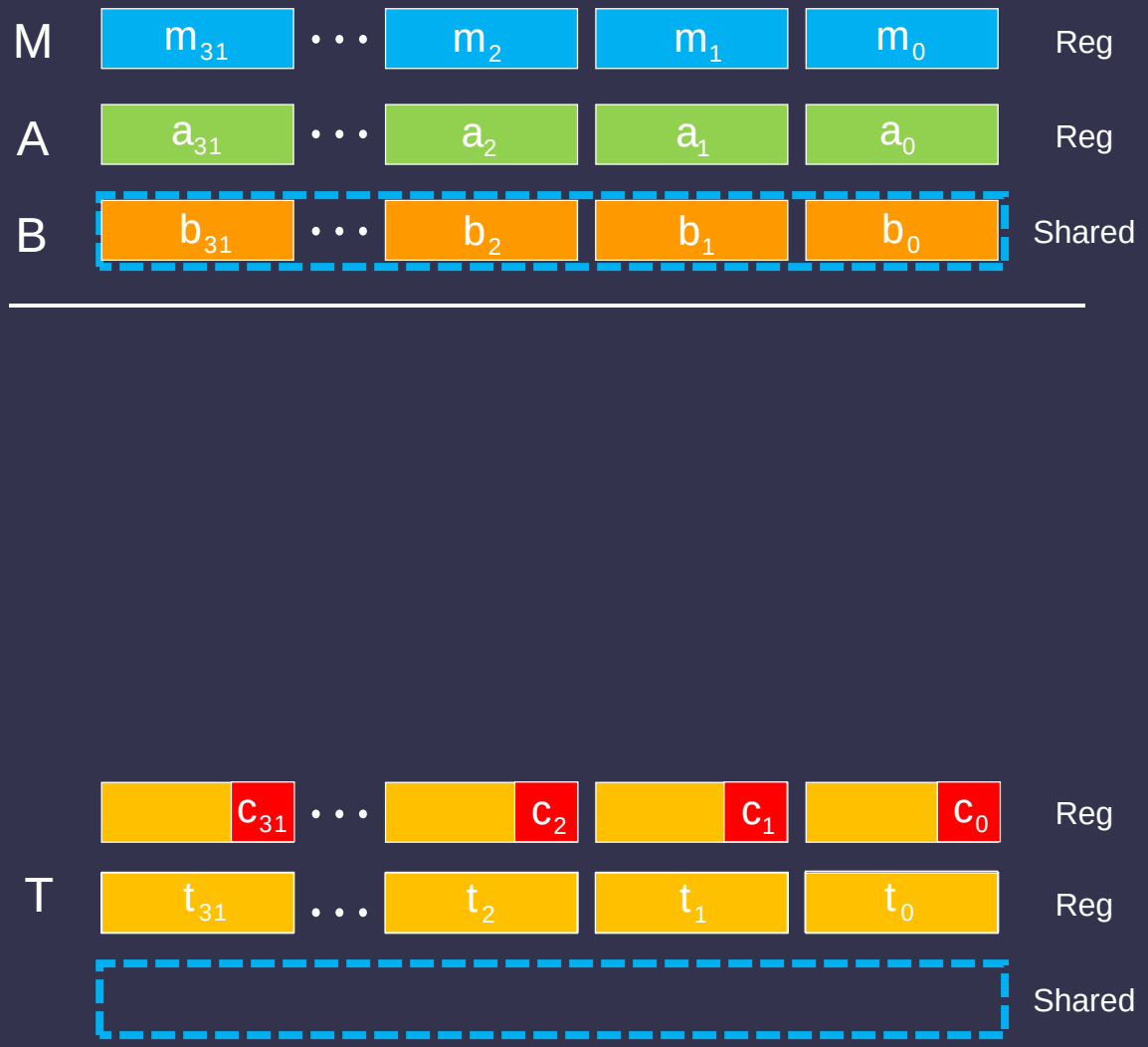


Montgomery Multiplication

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

Algorithm

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

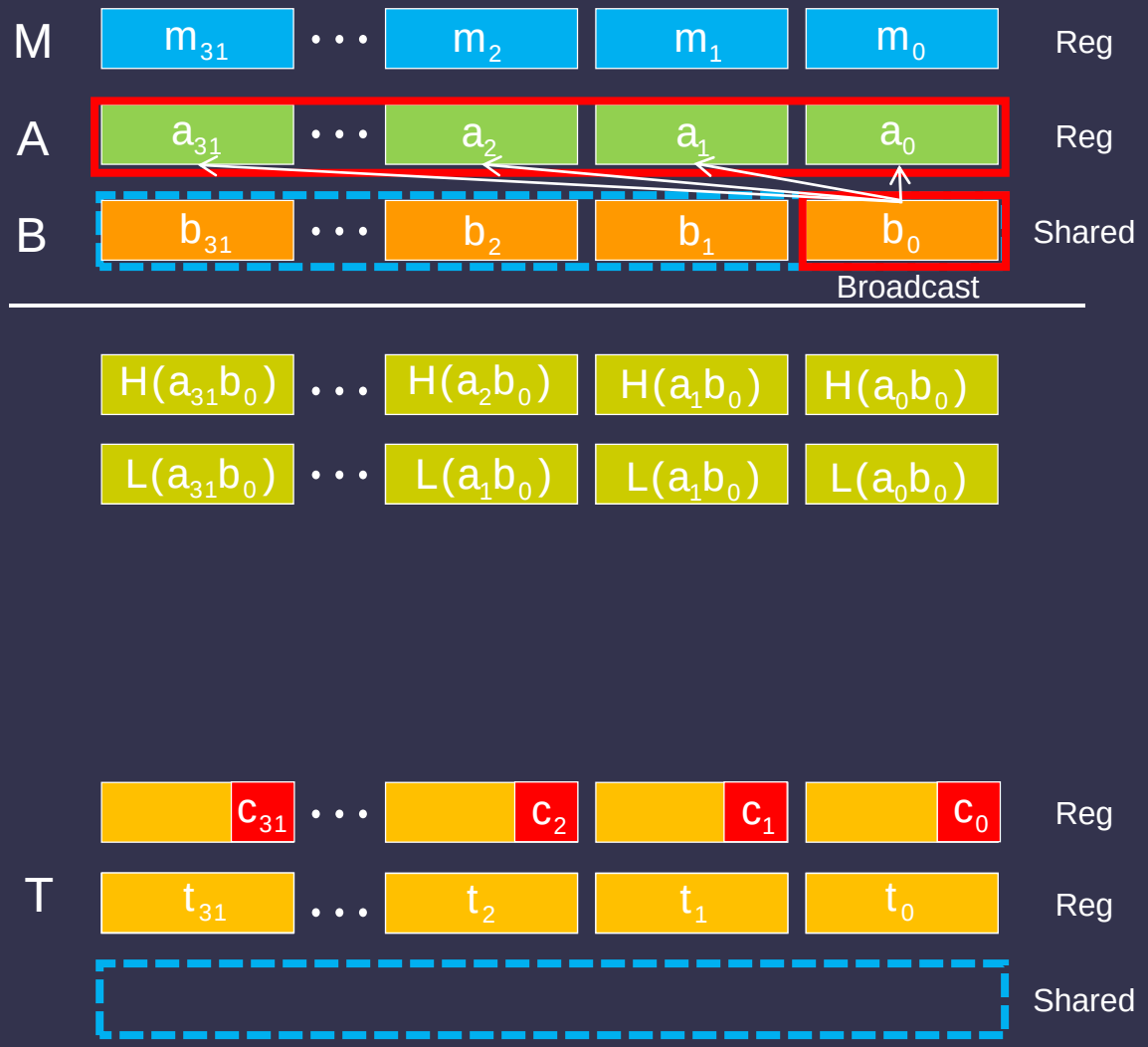


Montgomery Multiplication

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

Algorithm

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

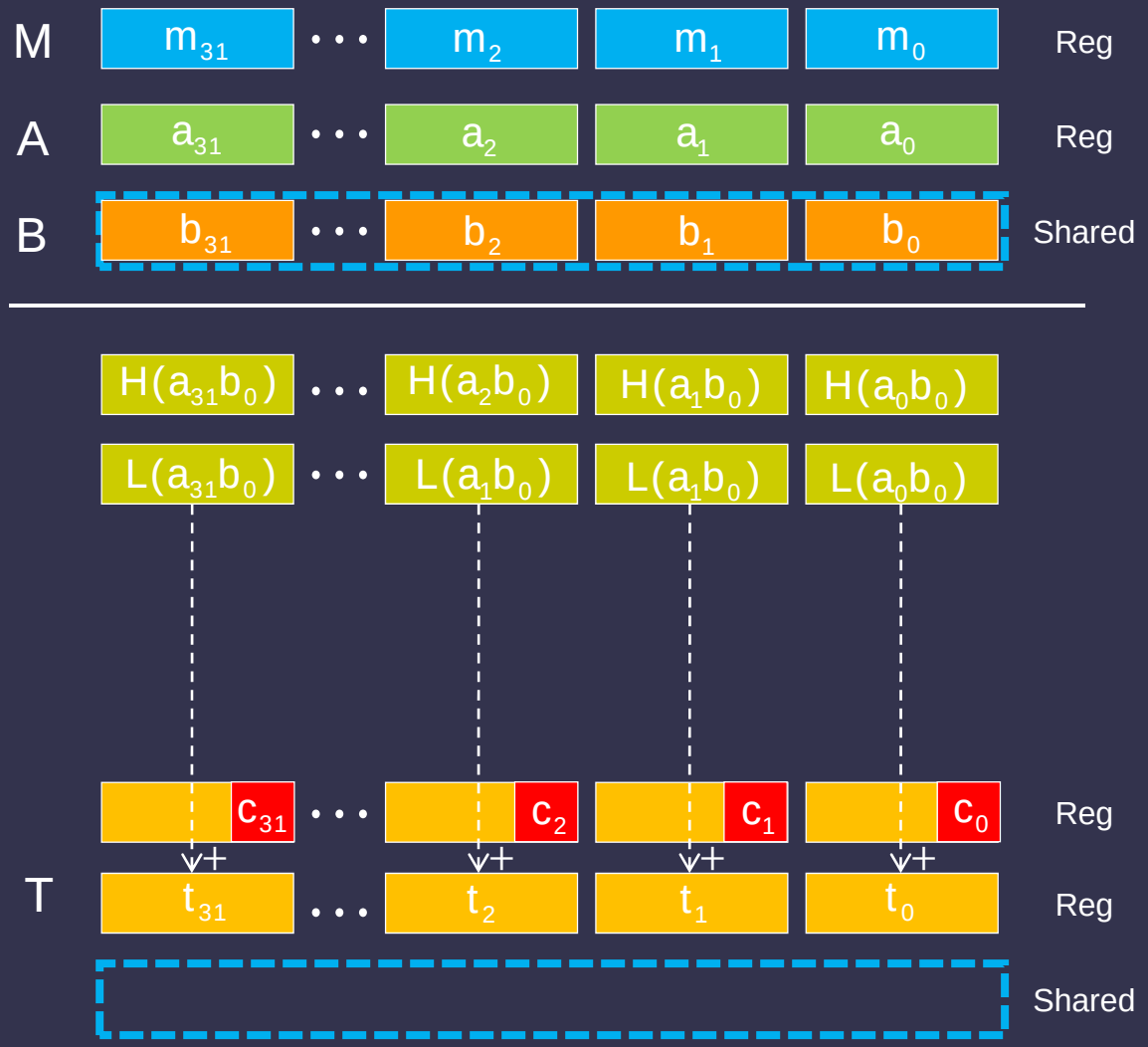


Montgomery Multiplication

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

Algorithm

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

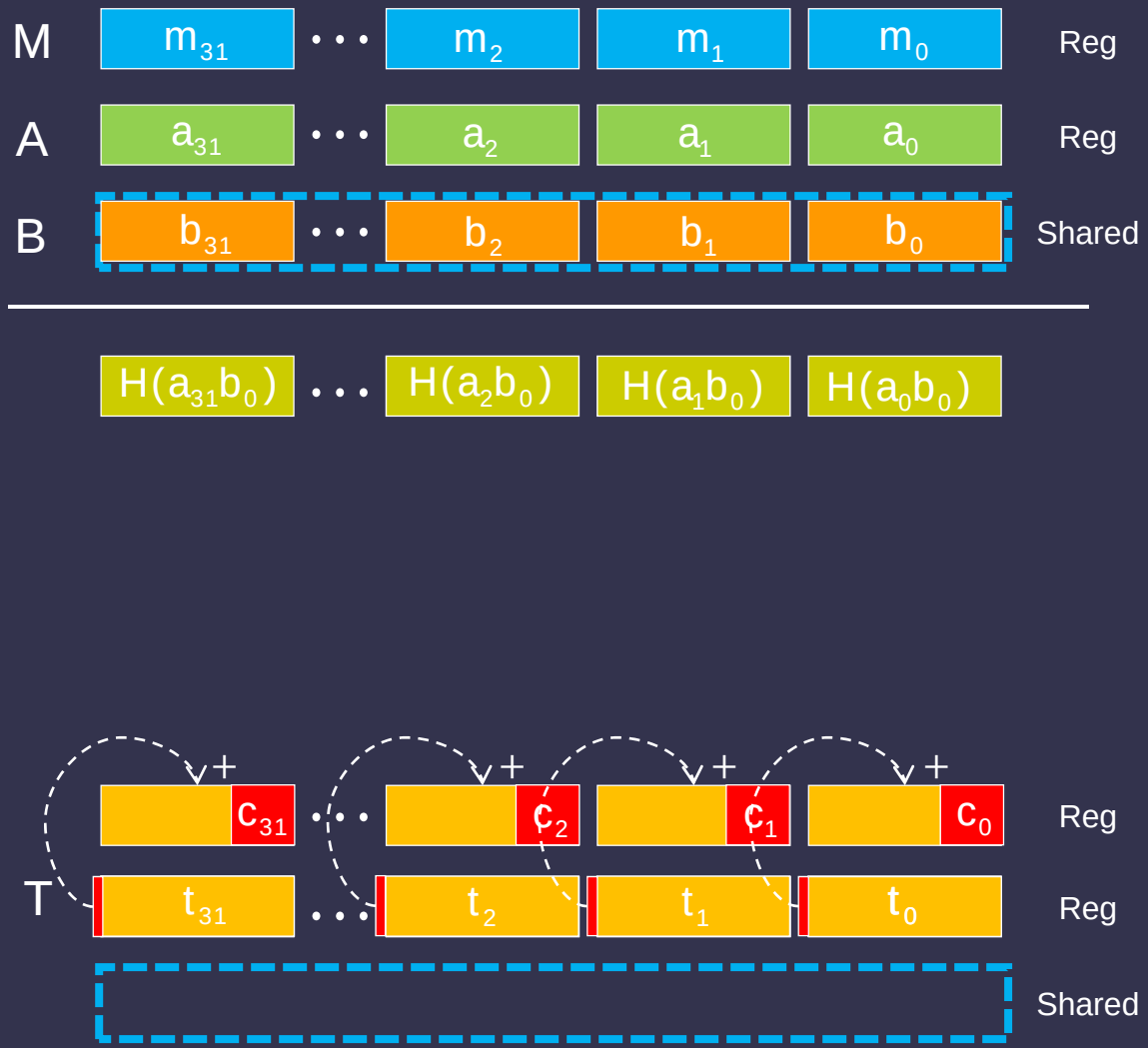


Montgomery Multiplication

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

Algorithm

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

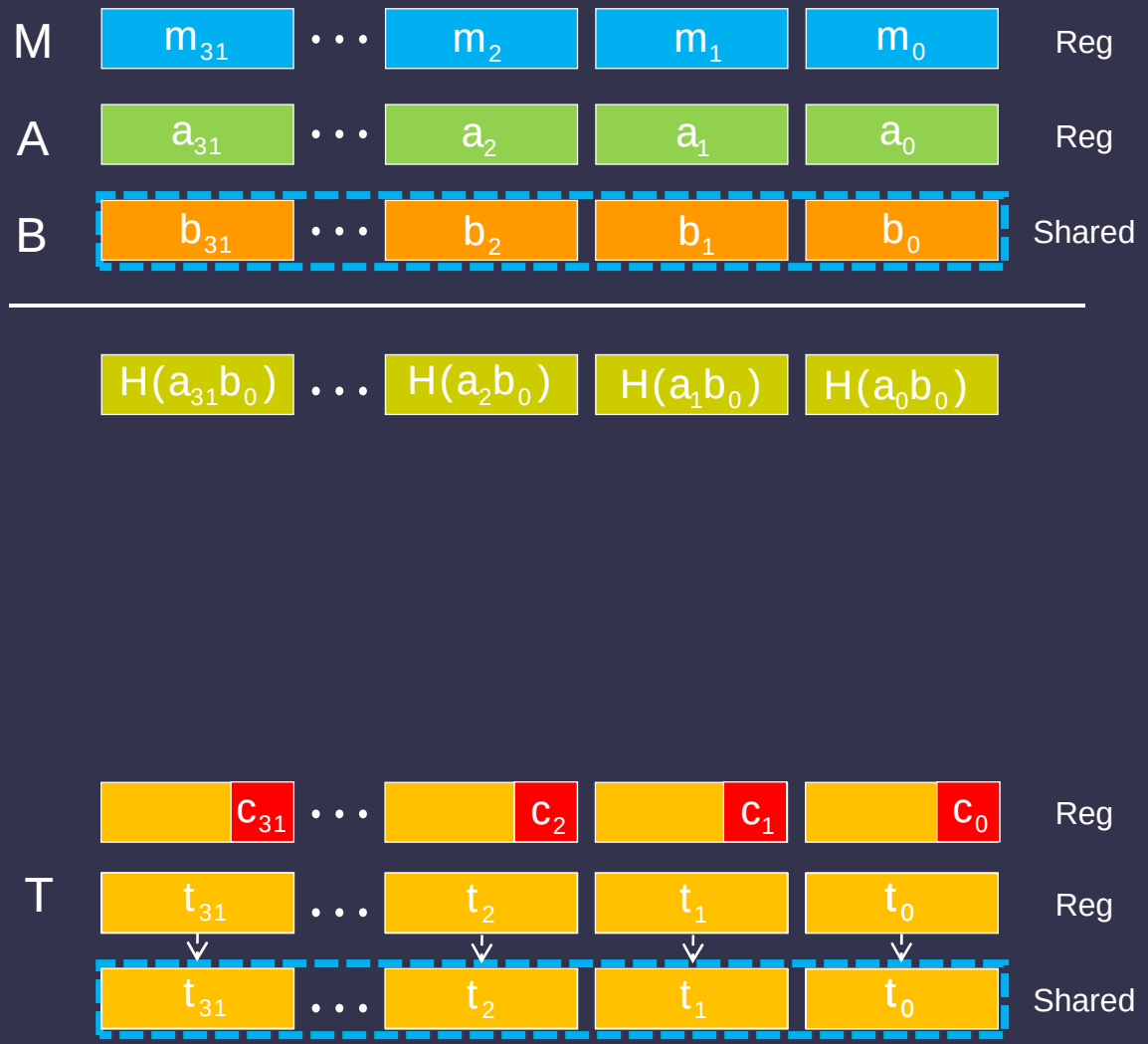


Montgomery Multiplication

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

Algorithm

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

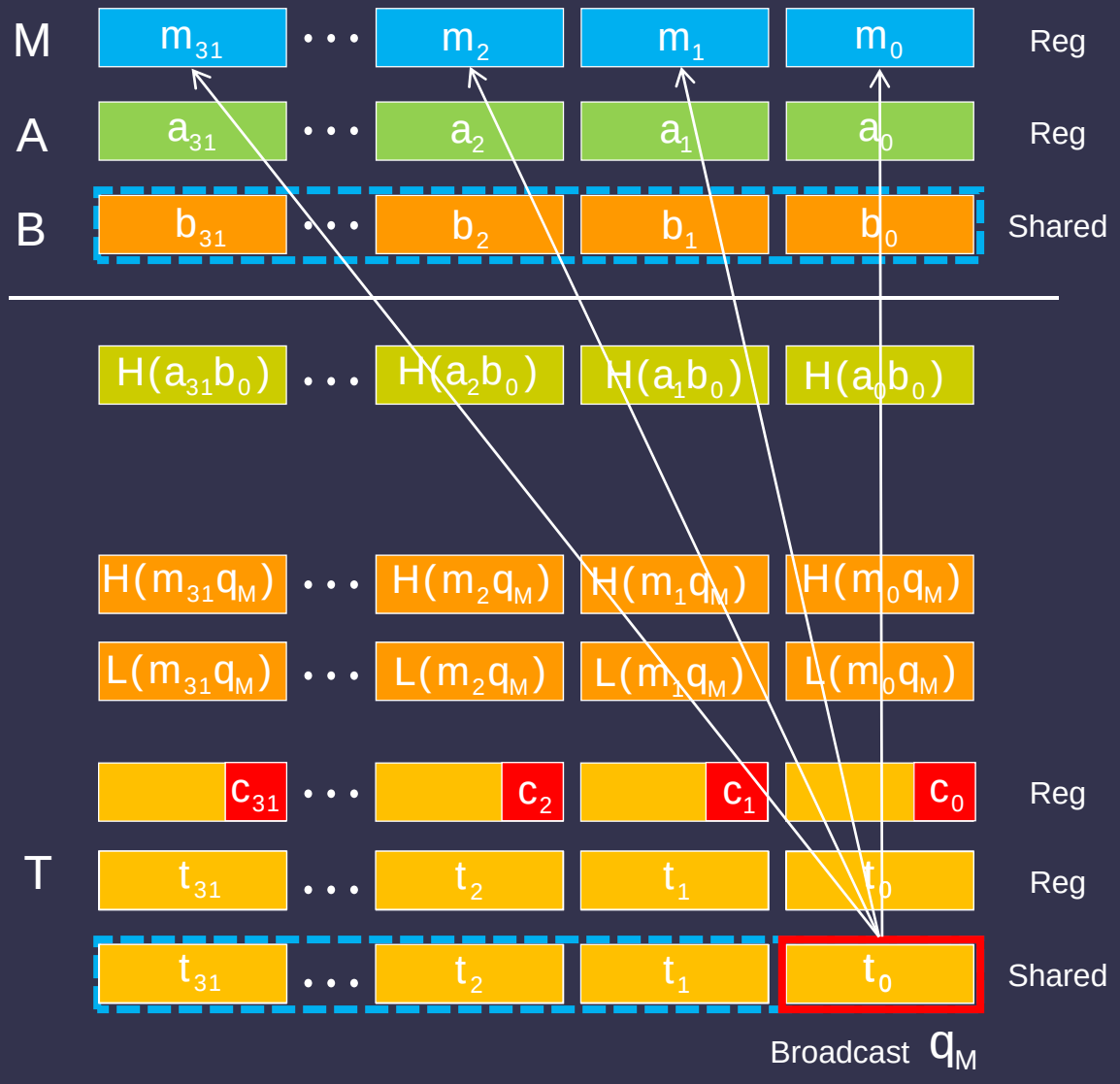


Montgomery Multiplication

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

Algorithm

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

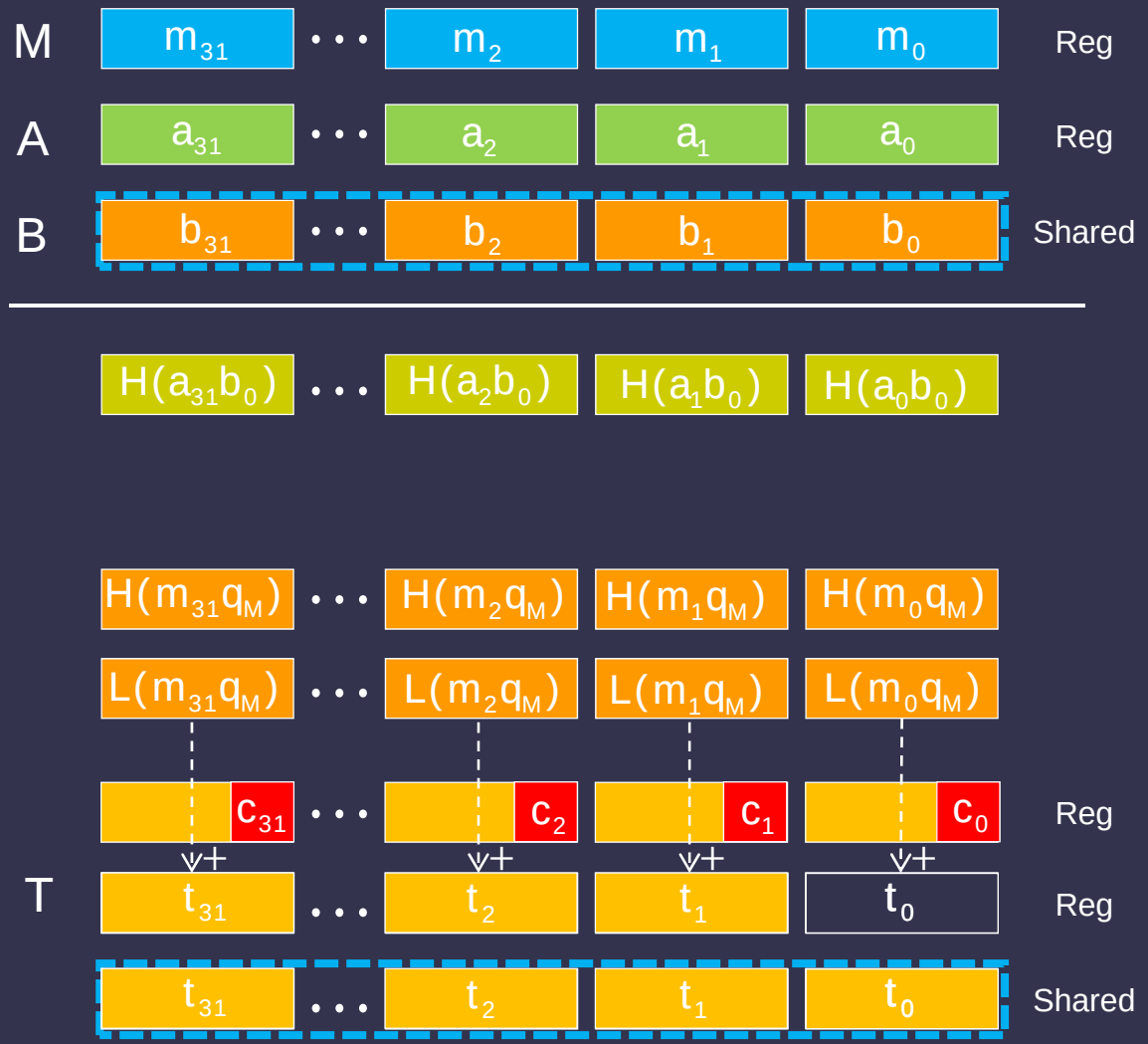


Montgomery Multiplication

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

Algorithm

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

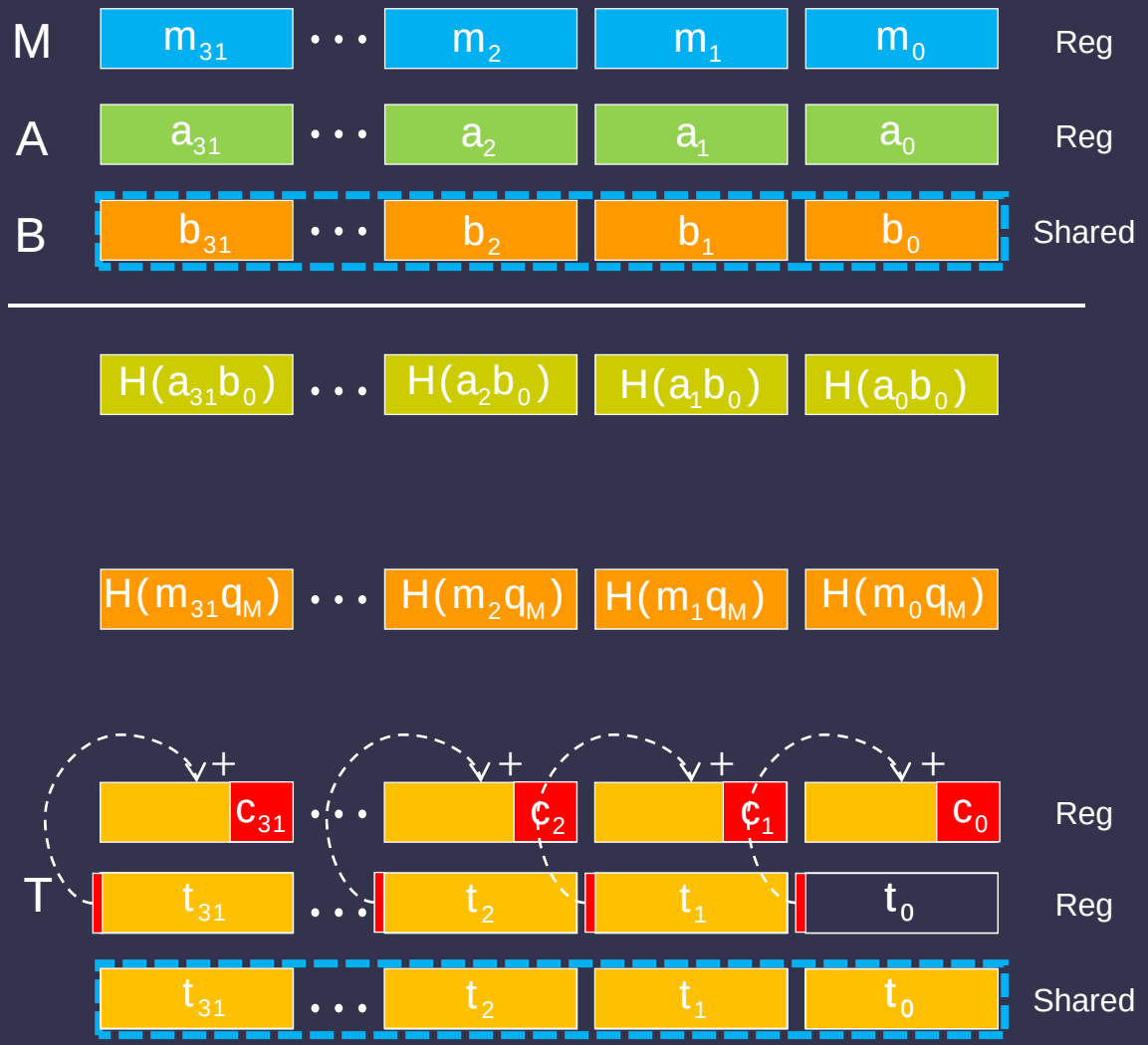


Montgomery Multiplication

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

Algorithm

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

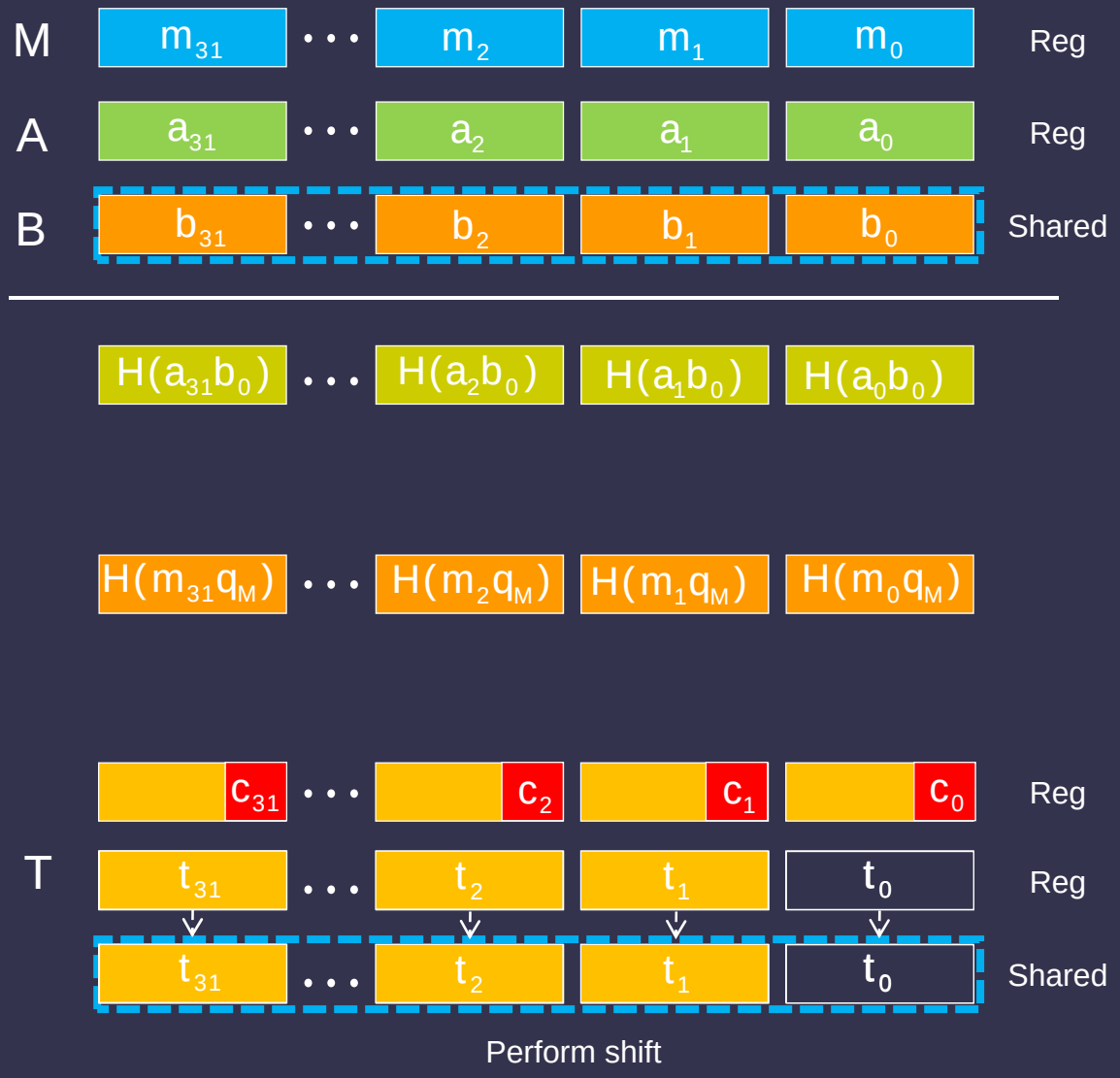


Montgomery Multiplication

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

Algorithm

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

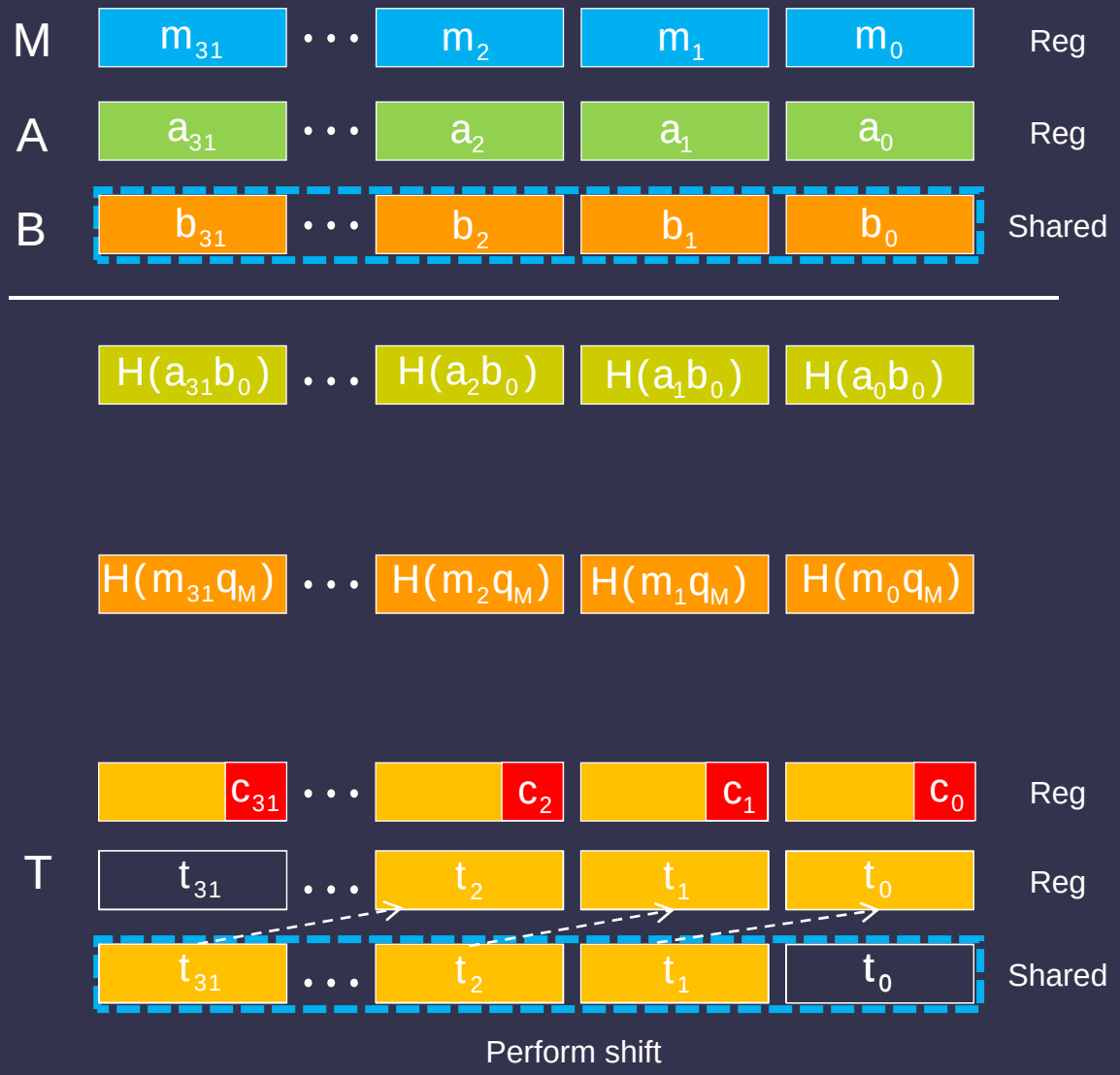


Montgomery Multiplication

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

Algorithm

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

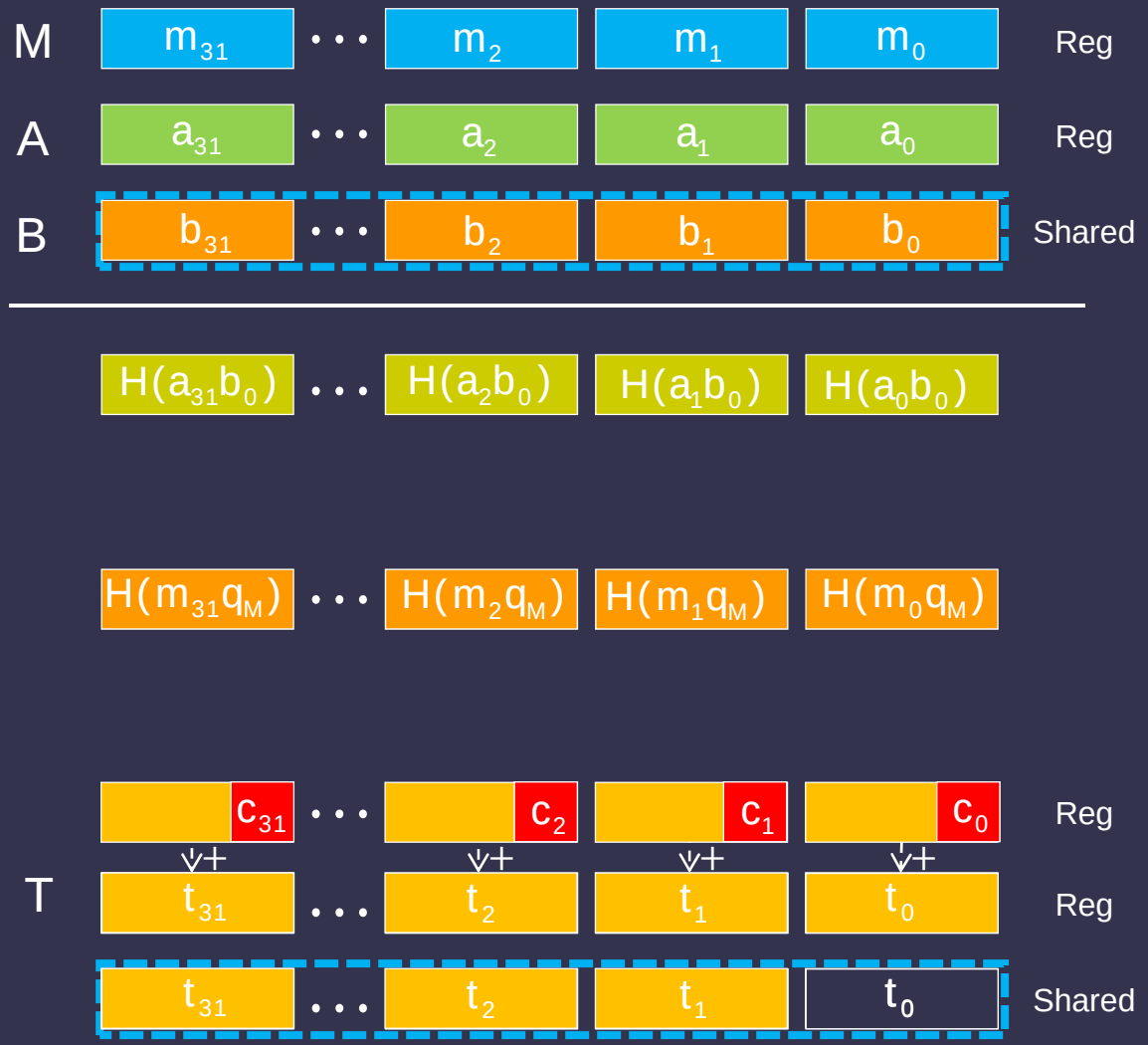


Montgomery Multiplication

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

Algorithm

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

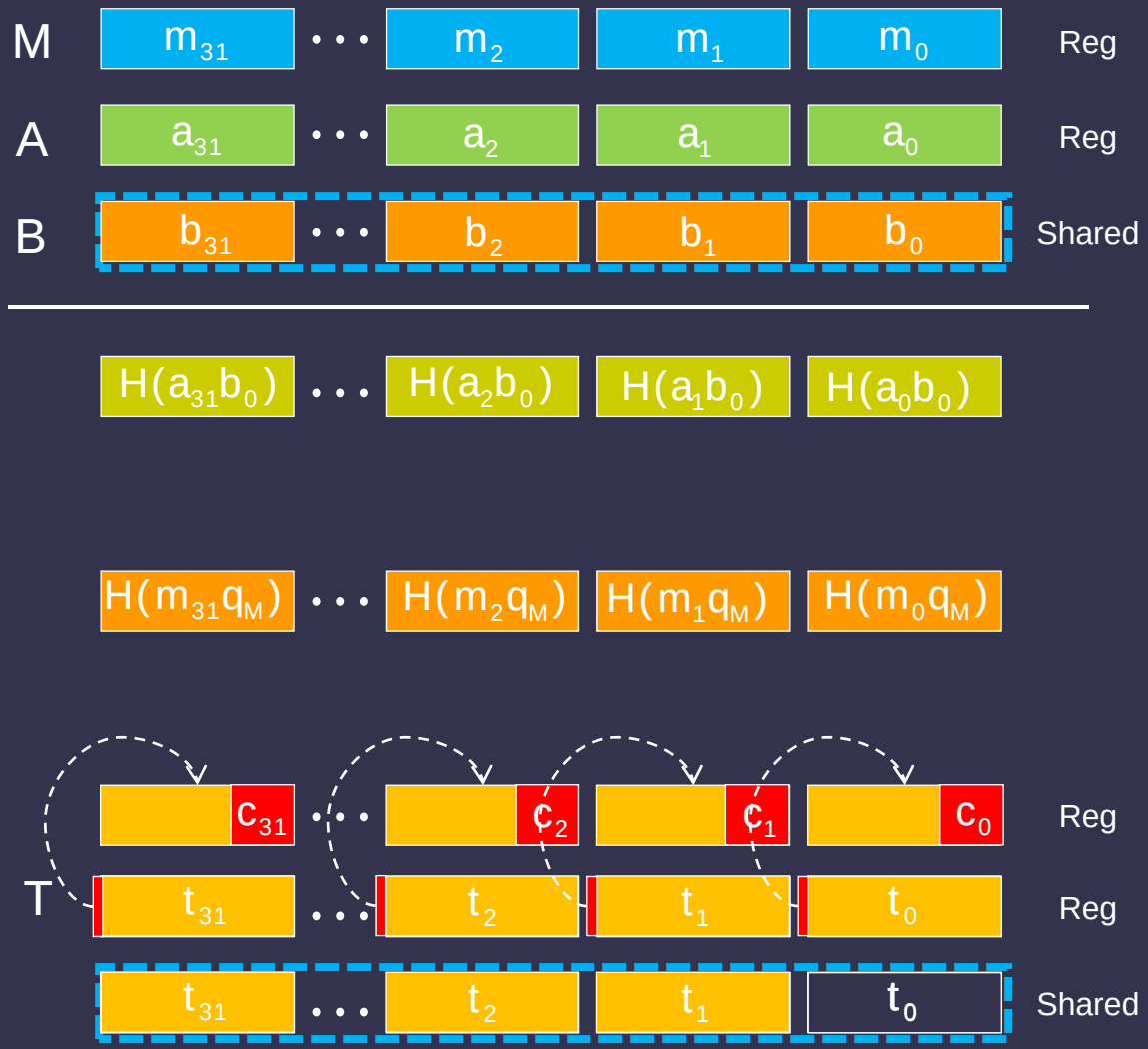


Montgomery Multiplication

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

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;
```

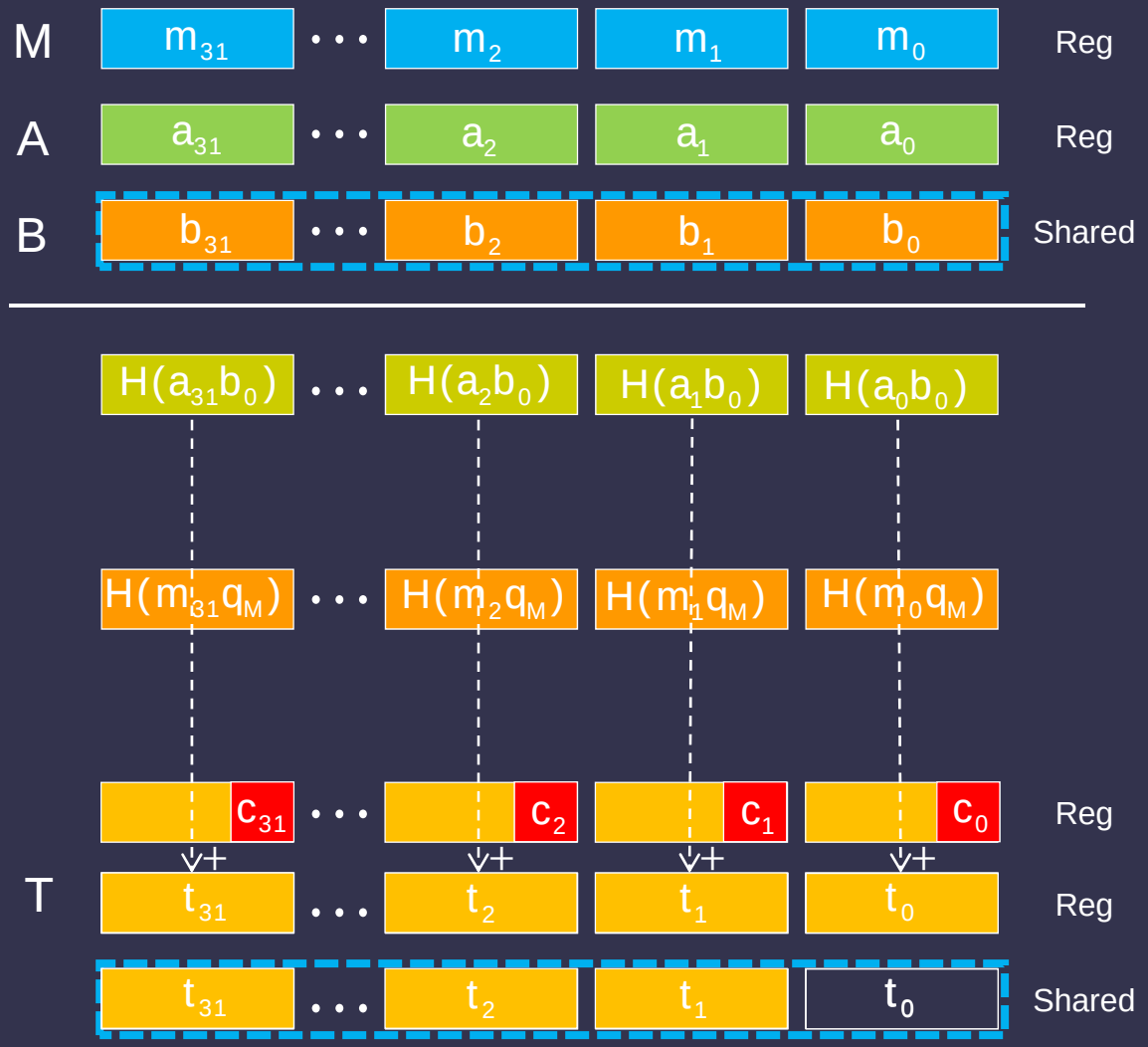


Montgomery Multiplication

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

Algorithm

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

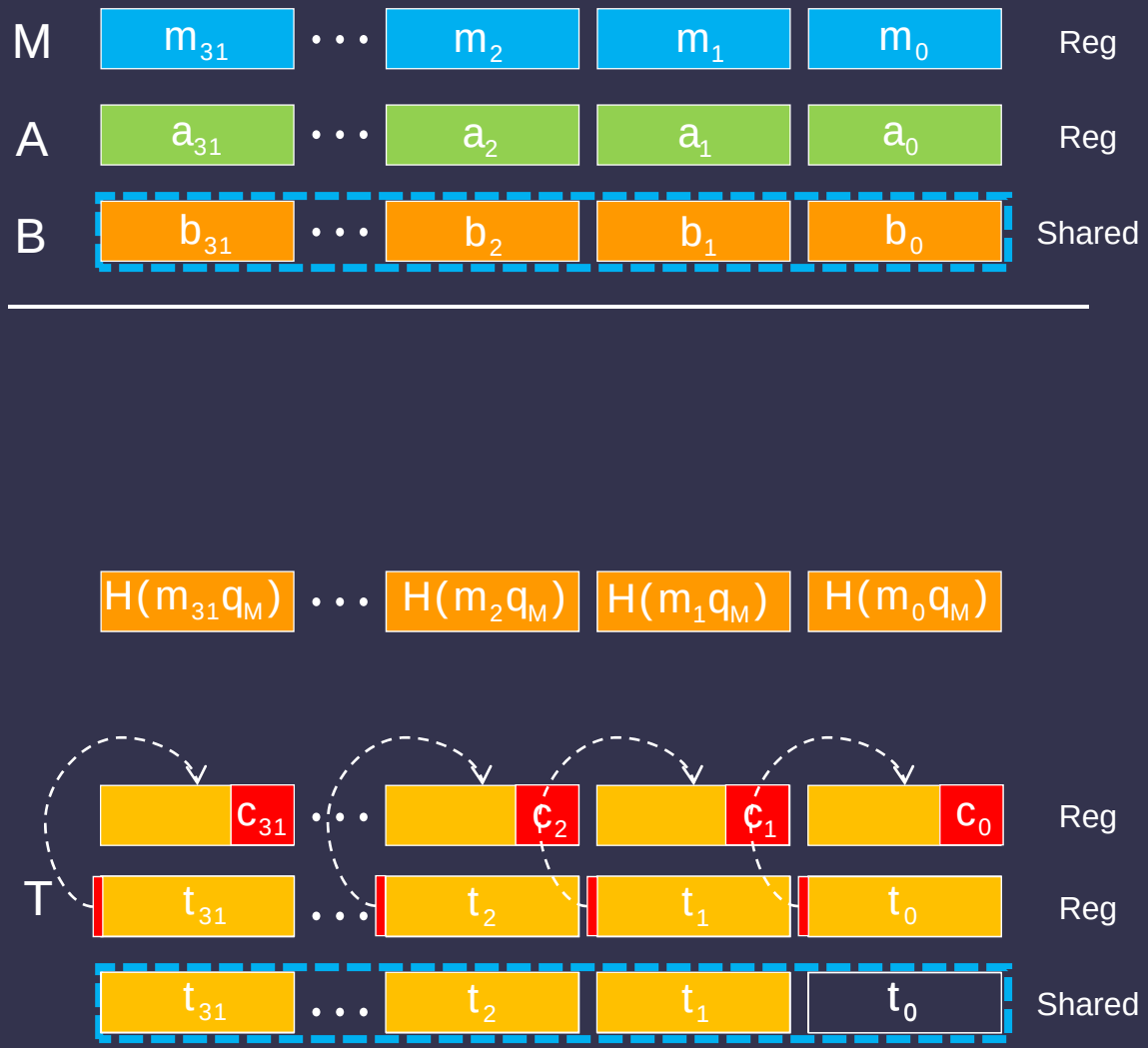


Montgomery Multiplication

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

Algorithm

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

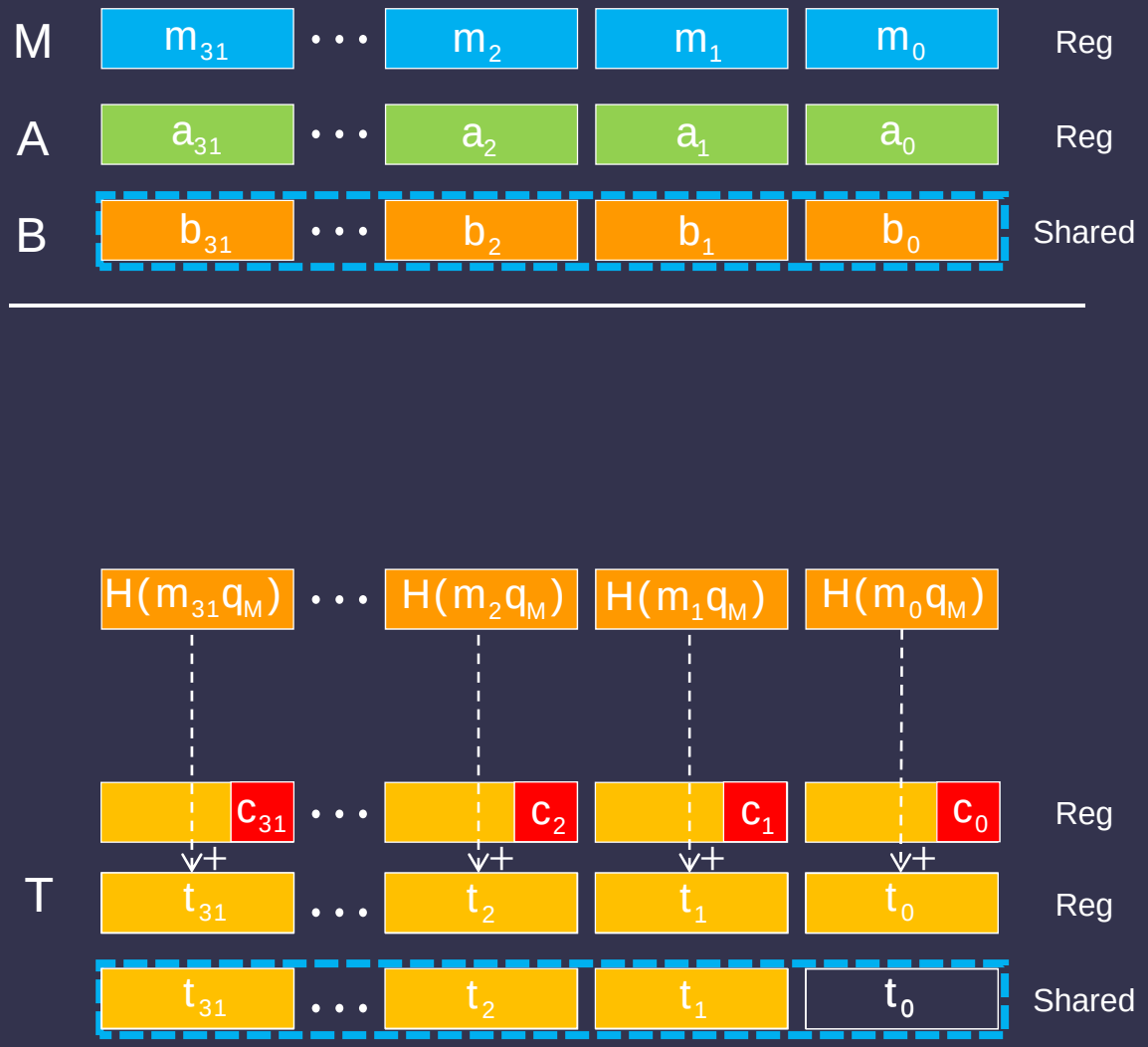


Montgomery Multiplication

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

Algorithm

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

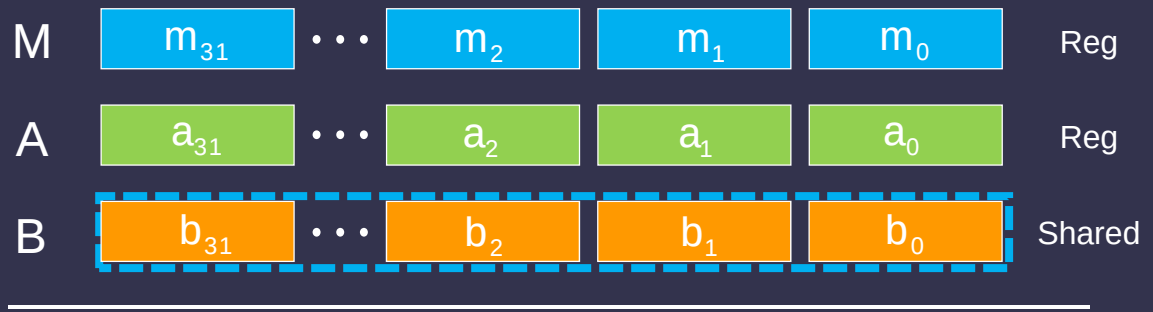


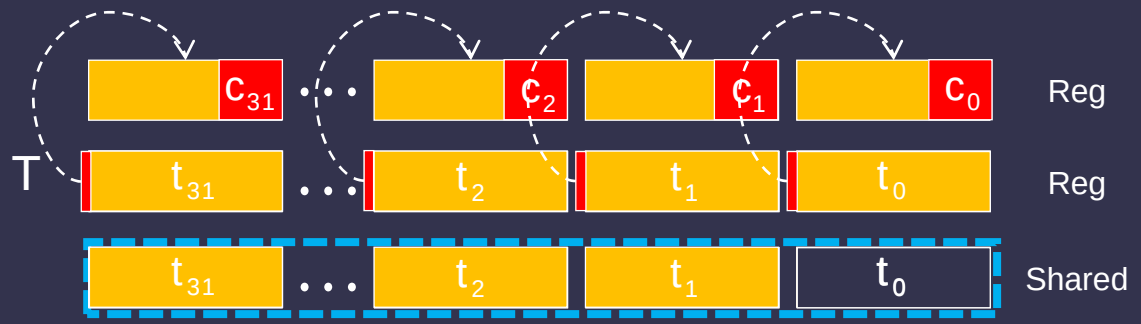
Montgomery Multiplication

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

Algorithm

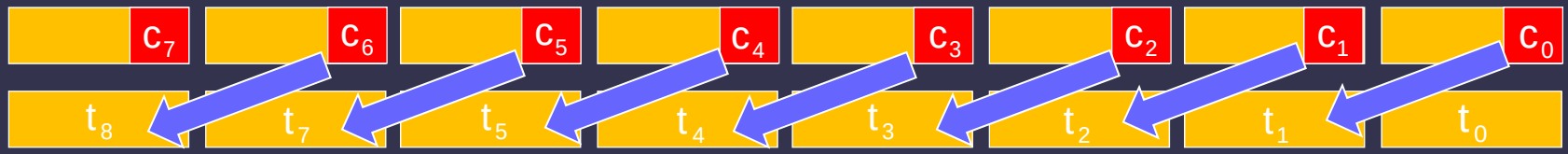
$A := \tilde{u}; B := \tilde{v}; M := m;$
$T := 0;$
for $(i = 0; i \leq s - 1; i++)$
{
  $T := T + b_i A;$
  $q_M := t_0 \cdot (-m_0)^{-1} \bmod B;$
  $T := (T + q_M \cdot M) \operatorname{div} B;$
}
if $T \geq M$ then $Z := T - M;$
else $Z := T;$





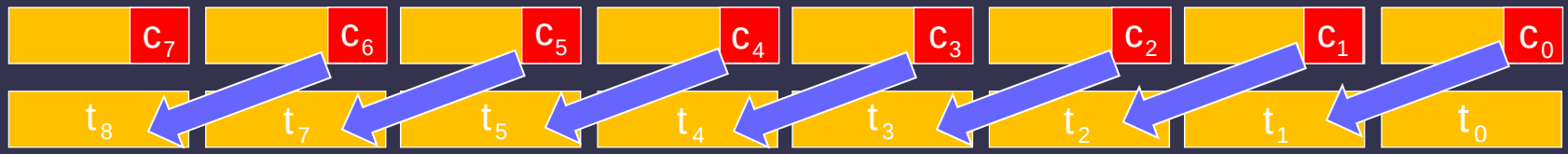
Avoiding carry propagation

- Requires 31 iteration of additions with carry



Avoiding carry propagation

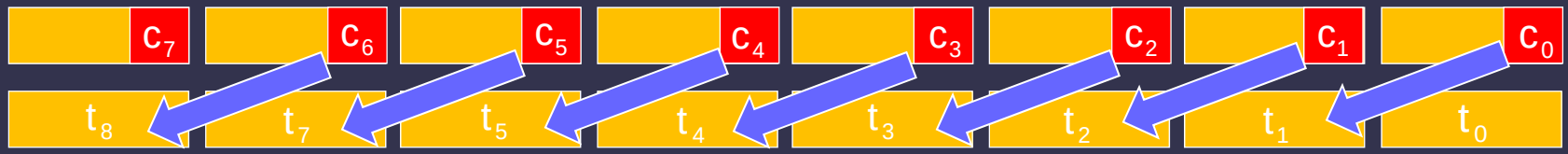
- Requires 31 iteration of additions with carry



- Very High Probability carries absorbed after one iteration.

Avoiding carry propagation

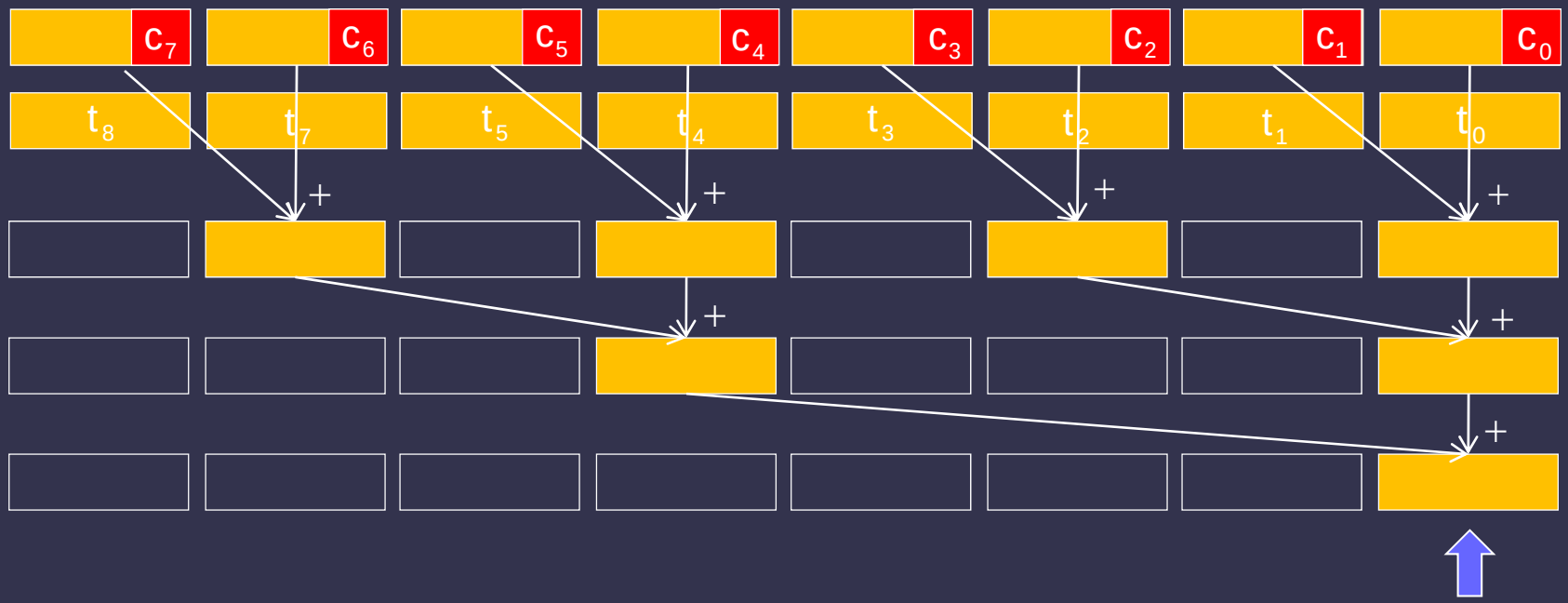
- Requires 31 iteration of additions with carry



- Very High Probability carries absorbed after one iteration.
- One time carry propagation + logarithmic time verification.

Avoiding carry propagation

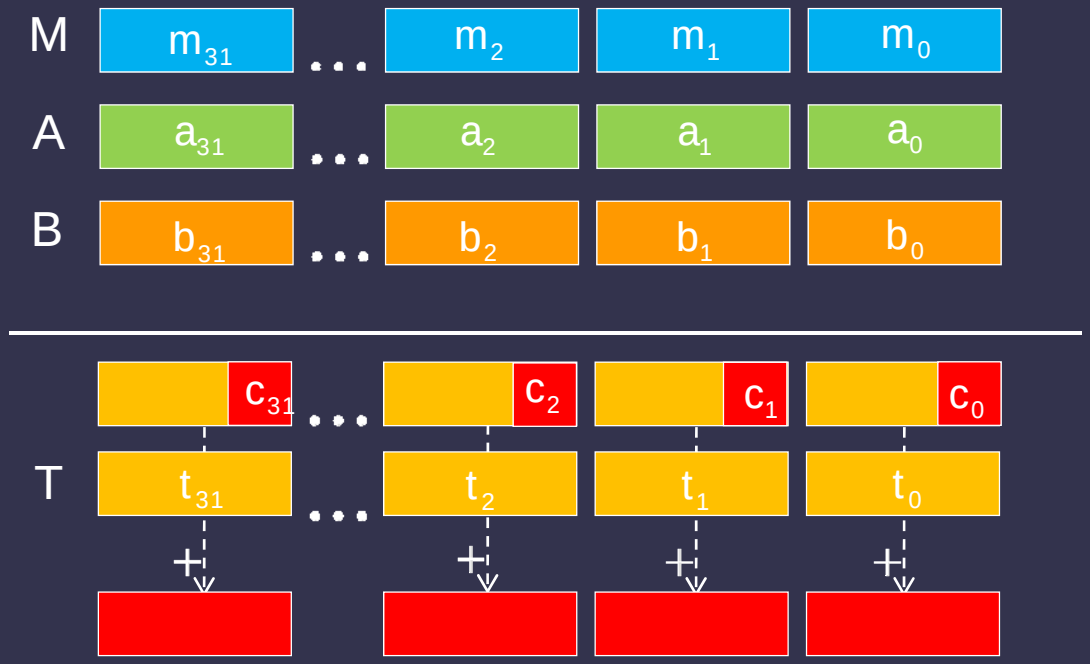
- Requires 31 iteration of additions with carry



- Very High Probability carries absorbed after one iteration.
- One time carry propagation + logarithmic time verification.

Speeding up further

- Prepare two exponentiation algorithms:
  1) *Fast Exp:* Accumulate remaining carries, check afterwards
  2) *Safe Exp:* Checks carries after every mod mul



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

Further optimization

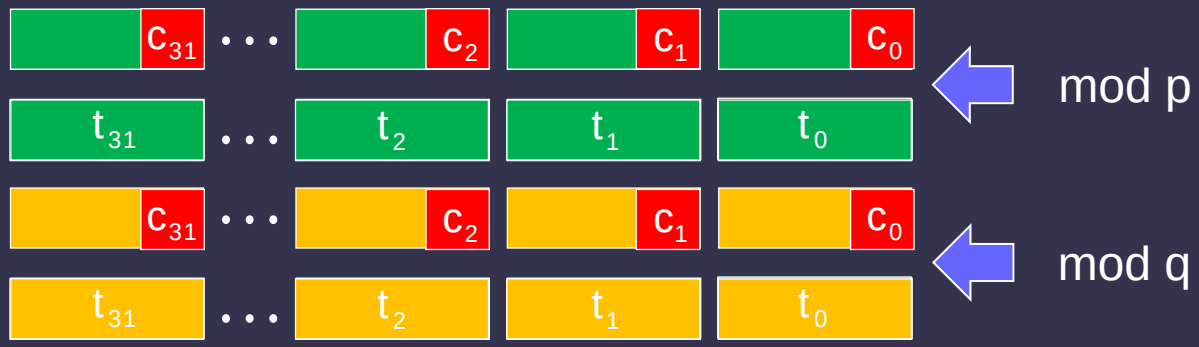
- *Use windows exponentiation with size w=5 bits*
- *Use inline assembly for Addition with Carry in CUDA*

Code:

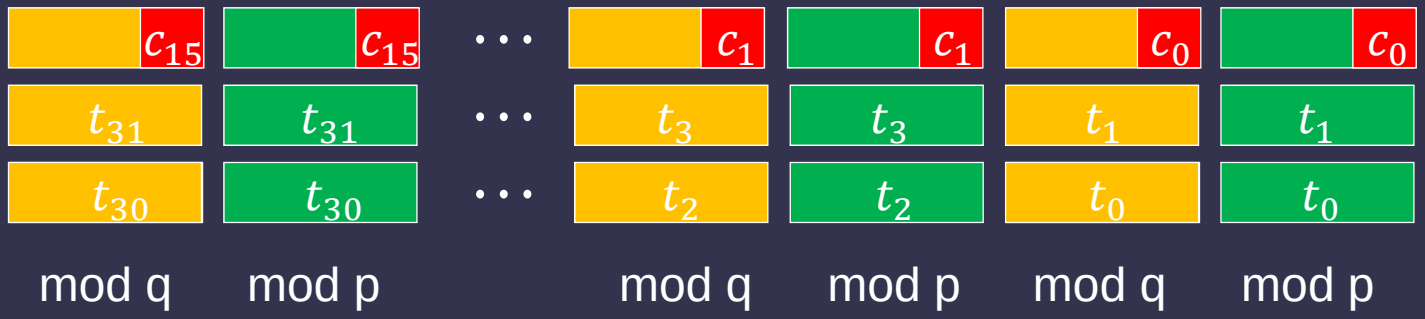
```
__device__ uint addc(uint a, uint b) {
  uint c;
  asm("addc.u32 %0, %1, %2;" : "=r" (c) : "r" (a) , "r" (b));
  return c;
}
__device__ uint addc_cc(uint a, uint b) {
  uint c;
  asm("addc.cc.u32 %0, %1, %2;" : "=r" (c) : "r" (a) , "r" (b));
  return c;
}
__device__ uint add_cc(uint a, uint b) {
  uint c;
  asm("add.cc.u32 %0, %1, %2;" : "=r" (c) : "r" (a) , "r" (b));
  return c;
}
```

Reducing carry propagation

- Need two set of data to process data moduli p and q
  (warps can be processed in parallel or sequentially)
- Try to process together inside one warp.

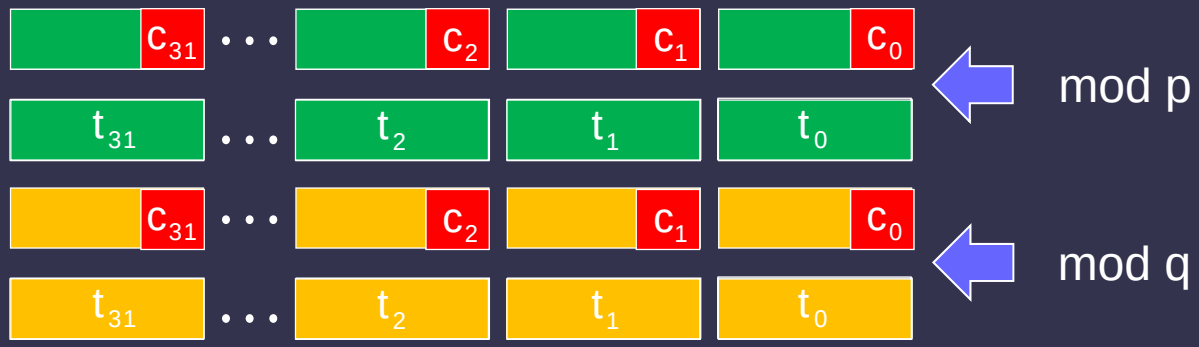


- Interleave operands mod p and mod q inside the warp.
- Use radix 2^{64}. Saves carry propagation.

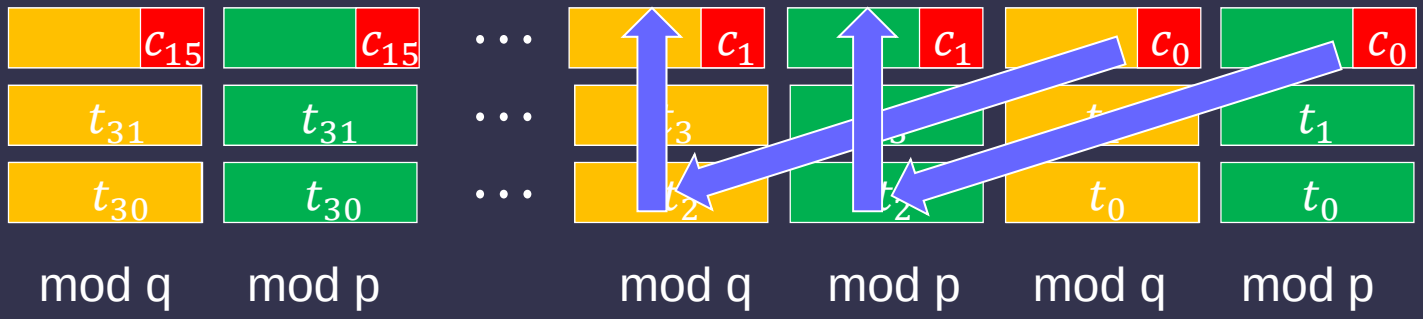


Reducing carry propagation

- Need two set of data to process data moduli p and q
  (warps can be processed in parallel or sequentially)
- Try to process together inside one warp.



- Interleave operands mod p and mod q inside the warp.
- Use radix 2^{64}. Saves carry propagation.

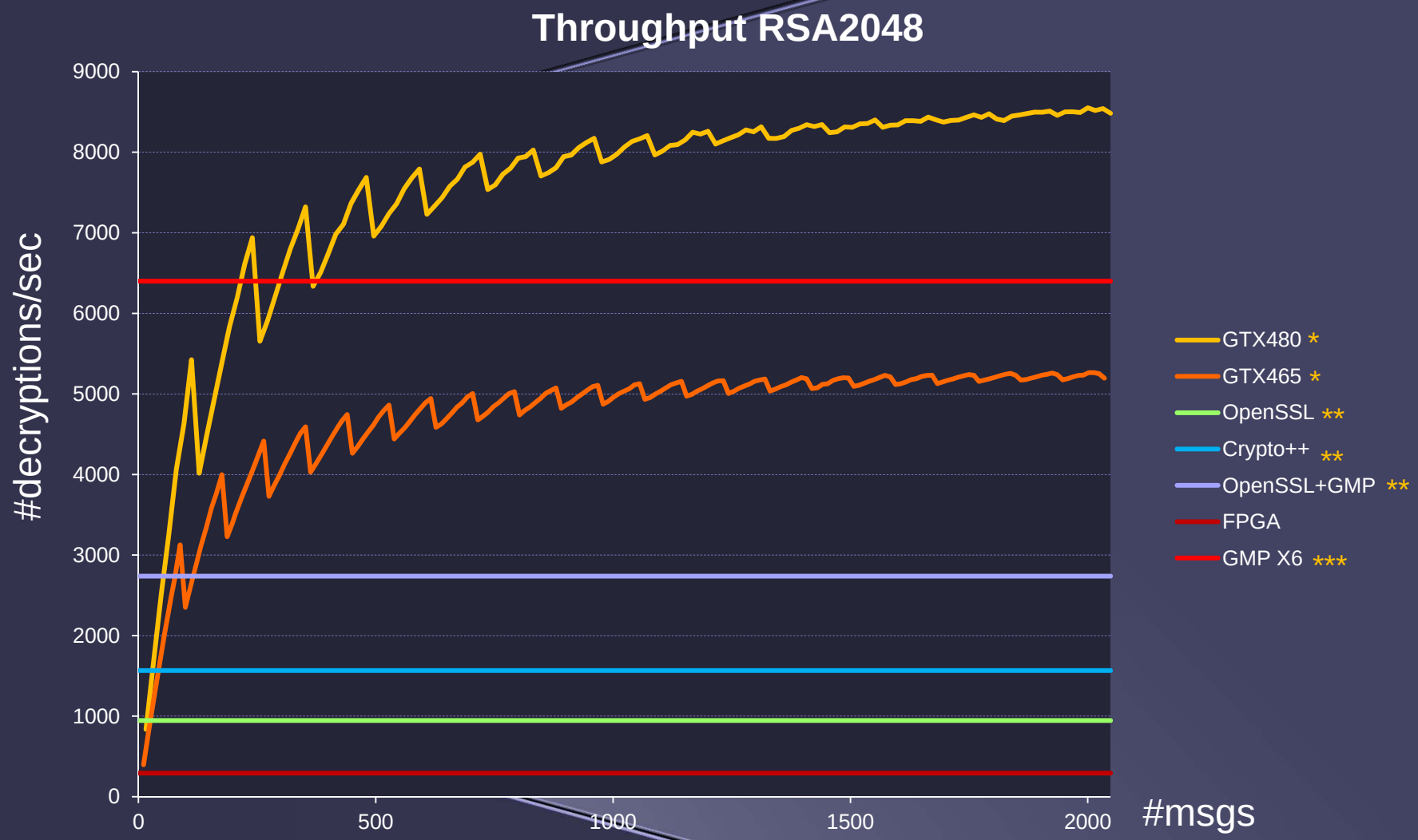


Performance Evaluation

GPU	NVIDIA GTX 480
#SM	16
Total #CUDA Cores	512
Device Clock Freq.	1’401 MHz
CUDA Kit	CUDA Tookit 3.2
CPU	Intel i7 960 Quad-Core @3.2GHz
OS	Debian Linux - 2.6.26-25

GPU	NVIDIA GTX 465
#SM	11
Total #CUDA Cores	352
Device Clock Freq.	1’215 MHz
CUDA Kit	CUDA Tookit 3.2
CPU	AMD Phenom 9500 Quad-Core @2.2GHz
OS	Ubuntu Linux 10.04 - 2.6.32-25

Performance Evaluation



* Includes I/O operands, CRT on CPU

** AMD Opteron™ 1381 Quad-Core @ 2.6GHz, GMP 5.0.1

*** AMD Phenom X6@3.2Ghz, GMP 5.0.1

Performance Evaluation

* Includes I/O operands, CRT on CPU

** AMD Opteron™ 1381 Quad-Core @ 2.6GHz

Performance Evaluation

Platform	Delay [ms]	Throughput [ops/sec] (decryptions)
OpenSSL Regular[1]	-	946
Crypto++[2]	-	1’566
OpenSSL + GMP[1]	-	2’738
GMP X6@3.2GHz	-	~6’400
FPGA XC4VFX12-10[4] 3937 slices, 17 DSP 48s	2 x 1.71 = 3.42	292
8800GTS (CIOS) [3] 112 cores @1.5GHz	55’184	104
8800GTS (RNS) [3] 112 cores @1.5GHz	849	57
GTX 465	28.1	3’126
	59.8	4’413
	380	5’264
GTX 480	20.6	5’424
	48	7’321
	237.8	8’542

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

(2) Evaluated on AMD Opteron™ 8354 @ 2.2GHz on Linux . (Scaled to 2.6GHz)

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

(4) D. Suzuki, “How to Maximize the Potential of FPGA Resources for Modular Exponentiation”, CHES 2007

Summary

- We have developed an algorithm for modular multiplication suitable for GPU that takes advantage of data coherence inside the warp.
- Current implementation is cost effective and competitive compared to CPU implementations. Suitable for server applications with low latency.
- The use of GPUs as cryptographic accelerators seems to be viable.