Efficient Parallel RSA Decryption Algorithm for Many-core GPUs with CUDA

Yu-Shiang Lin, Chun-Yuan Lin, Der-Chyuan Lou
Department of Computer Science and Information Engineering
Chang Gung University
Taoyuan 333, Taiwan, ROC
coldfunction@gmail.com, {cyulin, dclou}@mail.cgu.edu.tw

Outline

- **RSA method**
- **Pollard's *p*-1 factorization**
- **Introduce GPU and CUDA**
- **GPU-based Pollard's *p*-1 Factorization Algorithm**
- **Build Custom Integer System (CIS)**
- **Analysis & Results**

RSA method

- Key Generation
- Select p, q — p, q both prime, $p \neq q$
- Calculate $n = p*q$
- Calculate $\Phi(n) = (p-1)*(q-1)$
- Select integer e — $\gcd(\Phi(n), e) = 1; 1 < e < \Phi(n)$
- Calculate d
- Public key — $KU=\{e, n\}$
- Private key — $KR=\{d, n\}$

Encryption:
$M < n$ (M is plaintext)
$C = M^e (\mathrm{mod}\ n)$

Decryption:
$M = C^d (\mathrm{mod}\ n)$

p-1 factorization

- In 1974y, Pollard's p-1 Factorization Method
- Based on the Fermat's little theorem
- It can be subdivided into independent iterations

$$a^{p-1} \equiv 1 \pmod{p}$$

$$\mathrm{K} = a^{p-1} - 1 \equiv 0 \pmod{p}$$

$$gcd(\mathrm{K}, \mathrm{N}) = p \qquad \mathrm{p\text{-}1} \mid \mathrm{m'}$$

$$\mathrm{a}^{\mathrm{m}'} - 1 = \mathrm{a}^{(\mathrm{p}-1)^{\mathrm{c}}} - 1 = 1^{\mathrm{c}} \equiv 0 \pmod{p}$$

$$gcd(\mathrm{a}^{\mathrm{m}'} - 1, \mathrm{N}) = p$$

CPU-p-1 factorization algorithm (CPFA)

//object: to find one factor of integer N
//Load *prime table* to main memory from disk

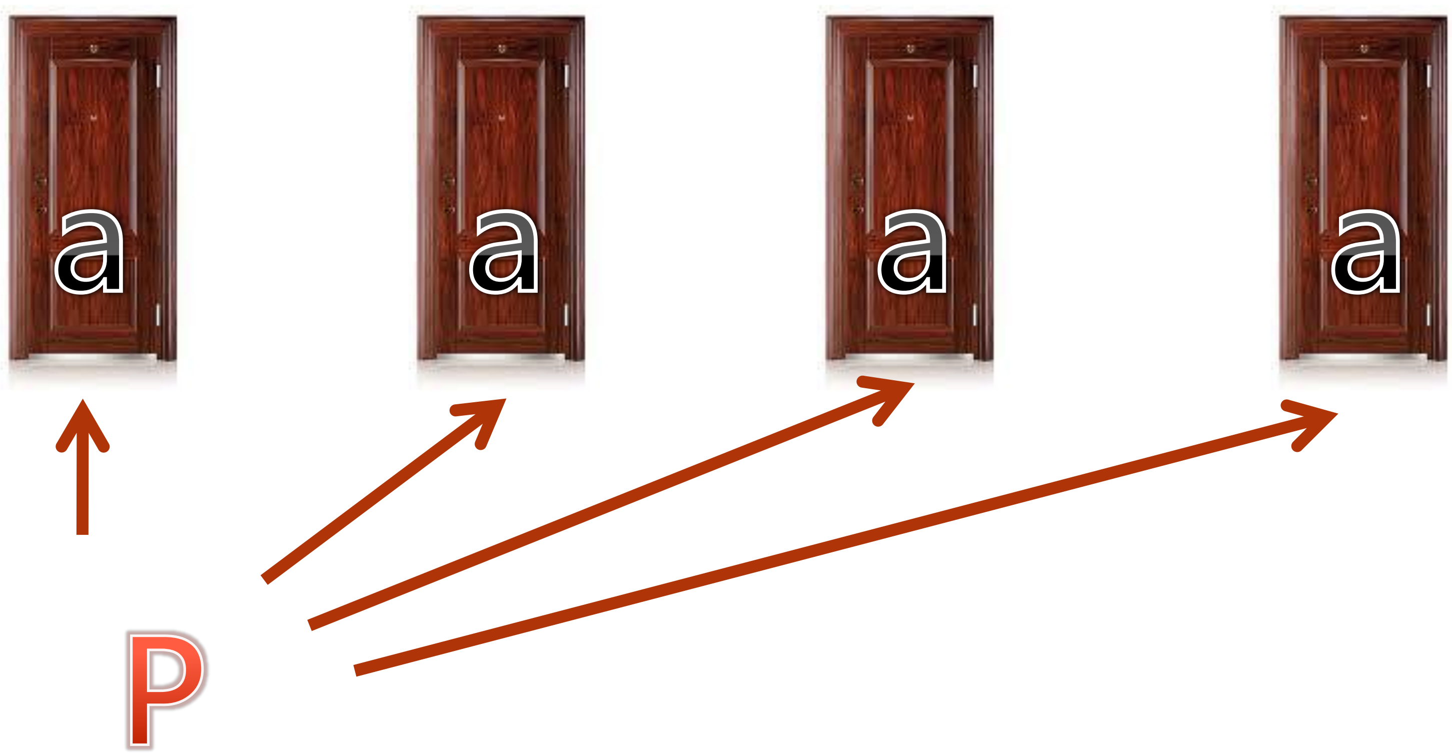
for (integer i from 1 to T_c)
{
1. Choose an integer a_c, it could be 2 or random generate.
2. Find prime p from *prime table* which are smaller than B.
3. Compute:

$$e = \prod_{\substack{\text{p prime} \\ 2 \leq p \leq B}} p^{\lfloor \log B / \log p \rfloor}$$

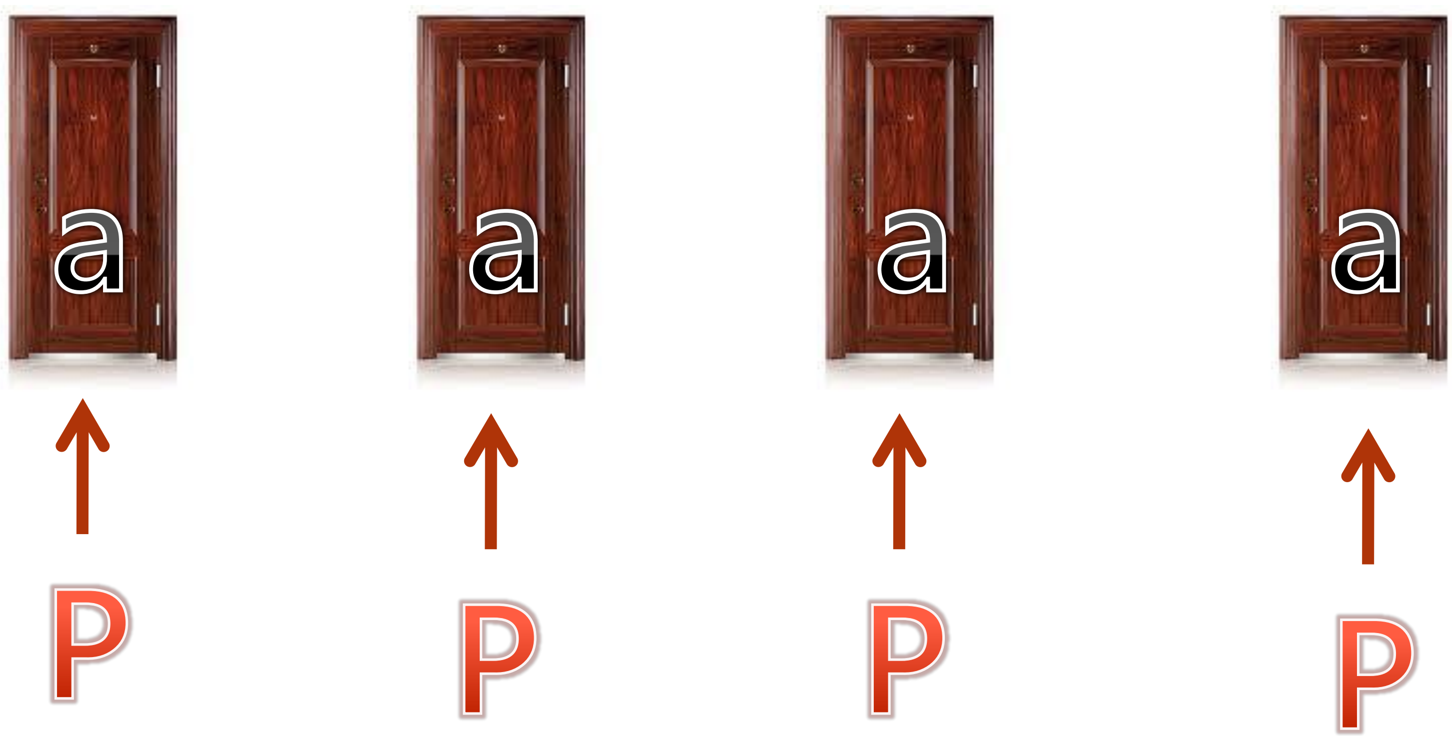
4. Let $b = a_c{}^e \bmod N$, if $1 < gcd(\text{b-1, n}) < N$, then return the value of greatest common divisor $gcd(b\text{-}1,N)$.
5. Follow step 4, if $gcd(b\text{-}1, N)$ equal 1 or N, then go to step 2.
6. If finding prime p over B, then execute the next iteration.


p
a

Sequential P-1



Parallel P-1



GPU

- SIMT
- Many cores
- Many kinds of memory
- More cheaper

CUDA

CUDA

Let GPU easy…

GPU-p-1 factorization algorithm (GPFA)

Inter-task parallelization

//object: to find one factor
//Load *prime table* from m

//gridDim.x is the built-in variable which represent the size of grid (number of blocks in one grid).
// blockDim.x is the built-in variable which represent the size of block (number of threads in one block).
//blockIdx.x is the built-in variable which represent the 1 D block index within the grid.
//threadIdx.x is the built-in variable which represent the 1 D thread index within the grid.

int linearID = blockDim.x*blockIdx.x+threadIdx.x;
int total_num_of_thread = gridDim.x*blockDim.x;
int a_g = 2+linearID;

for (integer i from blockIdx.x*blockDim.x to blockIdx.x*blockDim.x+T_g -1)
{

//pr(j) is the j-th prime number in the *prime table*.
for(unsigned int j= linearID; pr(j) < B ; j+= total_num_of_thread)
{

step 3 of CPU-*p*-1. Compute:

$$e = \prod_{\substack{\text{p prime} \\ 2 \le p \le B}} p^{\lfloor logB/logp \rfloor}$$

step 4. Let $b = a_g{}^e \bmod N$, if $1 < gcd(\text{b-1, n}) < N$, then return the value of greatest common divisor $gcd(b\text{-}1, N)$ to global memory.
step 5 of CPU-*p*-1. Follow step 4, if $gcd(b\text{-}1, N)$ equal 1 or N, then continue.
}
$a_g = a_g * a_g$ %RAND_MAX+i+linearID;
}

GPU-p-1 factorization algorithm (GPFA)

//object: to find one factor of integer N
//Load *prime table* from main memory of CPU to Global memory of GPU.

//gridDim.x is the built-in variable which represent the size of grid (number of blocks in one grid).
// blockDim.x is the built-in variable which represent the size of block (number of threads in one block).
//blockIdx.x is the built-in variable which represent the 1 D block index within the grid.
//threadIdx.x is the built-in variable which represent the 1 D thread index within the grid.

int linearID = blockDim.x*blockIdx.x+threadIdx.x;
int total_num_of_thread = gridDim.x*blockDim.x;
int a_g = 2+linearID;

for (integer i from blockIdx.x*blockDim.x to blockIdx.x*blockDim.x+T_g -1)
{

$a_g = a_g * a_g$ %RAND_MAX+i+linearID;

step 3 of CPU-p-1. Compute:

$$e = \prod_{\substack{\text{p prime} \\ 2 \le p \le B}} p^{\lfloor logB/logp \rfloor}$$

step 4. Let $b = a_g{}^e \bmod N$, if $1 < gcd(\text{b-1, n}) < N$, then return the value of greatest common divisor $gcd(b\text{-}1,N)$ to global memory.
step 5 of CPU-p-1. Follow step 4, if $gcd(b\text{-}1, N)$ equal 1 or N, then continue.
}
$a_g = a_g * a_g$ %RAND_MAX+i+linearID;
}

GPU memory allocate

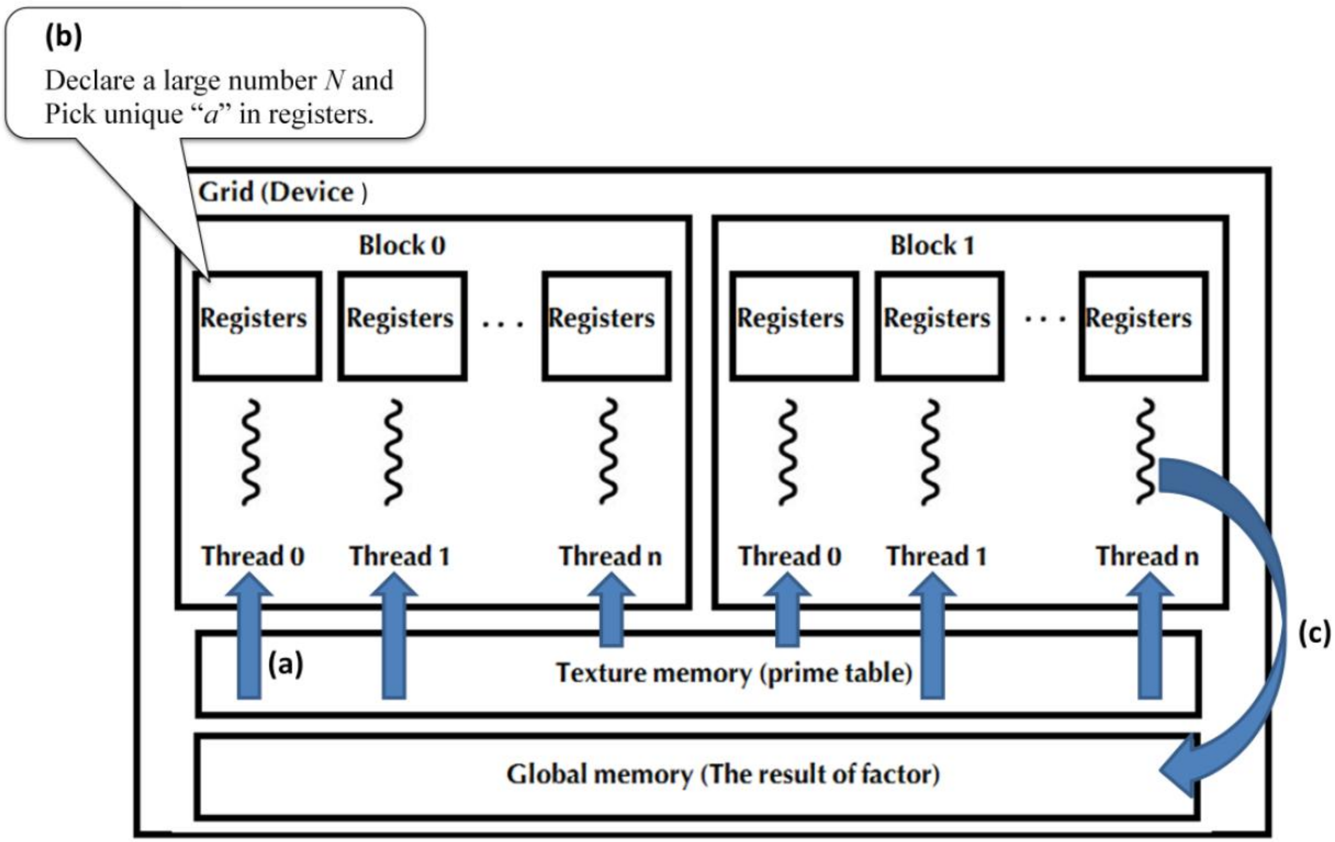


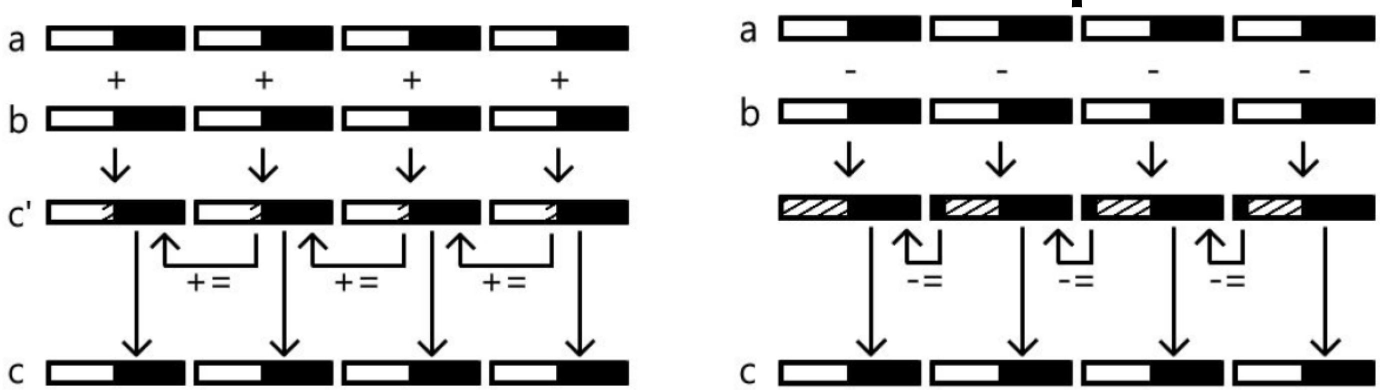
Figure 3. In CUDA, memory allocate scenario: (a)Pre-allocate *prime table* in texture memory before execute kernel function. (b) Store the large number N and a in registers. (c) Store the result to global memory.

CIS (Custom Integer System)

- **An example of integer representation in CIS**

Figure 1. *CSBI*: The black area is the first half of the unsigned integer, such as the storage space of a large integer (totally 128 bit), the white area is the latter half of unsigned integer, such as the additional carry binary space (totally 128 bit).

- **The addition and subtraction operations in CIS**



(a) large integer addition. (b) large integer subtraction.

Figure 2. (a): It take the sum of each element of a and b array, and temporarily store into c', then refresh the additional carry binary space added to the higher binary space sequentially. (b): The principle of large integer subtraction much like (a), it take the subtraction of each element of a and b array, if a[i] < b[i], then you need to borrow 1 from a[i+1].

Why use *CIS*

- **Our *GPFA* is applicable to *inter-task parallelization***
- **Unnecessary for the individual operator of large integers in parallel**
- **Mod operation cost is expensive per thread on GPU**

Test Environments

- **Test Platform**

CPU	**Intel Core2 Quad Q8200 2.33GHz**
Memory	**4 GB RAM**
GPU	**C2050, S1070, GTX-260**
OS	**Linux**

- **Bbenchmark**

	N	*p*	*q*
RSA-41	0x12B1F259795	0x721F7	0x29EFD3
RSA-44	0x89FD383381B	0x120FC7	0x7A3D0D
RSA-46	0x3CF5F89ED5F5	0xC50069	0x4F37AD
RSA-47	0x600FF385C031	0xFF52D9	0x605119
RSA-48	0x878D4C7D68E9	0xABB039	0xCA1E31
RSA-56	0xB8C8CBD2DAEE7D	0xD985797	0xD978D0B
RSA-64	0x6926C73F919FA3E7	0x79E6711B	0xDCD39125

Analysis & Results

$$speedup = \frac{Time(CPFA)}{Time(GPFA)} = \frac{(\frac{B}{\ln B}u)t_c}{(s\times(\frac{B}{\ln B\times(\text{gridDim.x}\times\text{blockDim.x})}))t_g}$$

$$= (\frac{u\times t_c}{s\times t_g})\times(\text{gridDim.x}\times\text{blockDim.x})$$

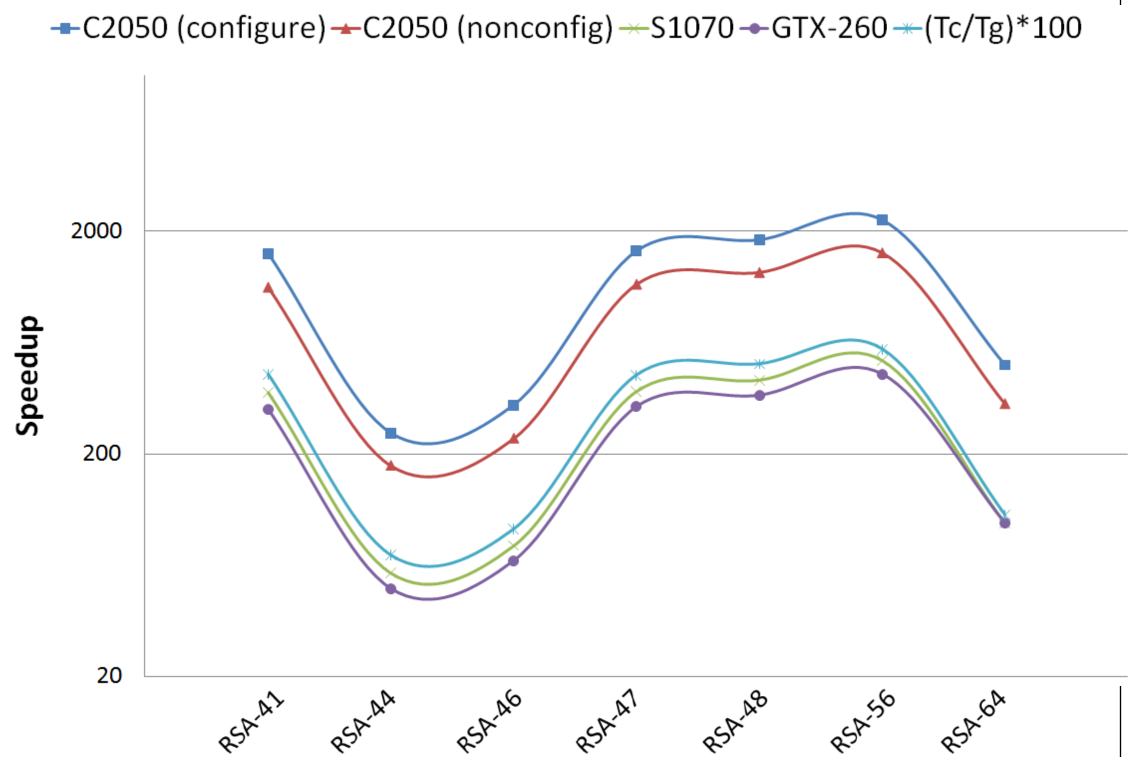


Figure 5. The speedup compare with *CPFA* in CPU, which test data from RSA-41 to RSA-64, and y-axis is the scale of logarithm to base 10.

Thank You

For Your

Listening…