Accelerating SSL by offloading RSA decryption to GPGPUs in a heterogeneous environment

Dissertation

submitted in partial fulfillment of the requirements
for the degree of

Master of Technology

by

Gaurav Gupta
Roll No: 2013MCS2556

under the guidance of

Prof. Subodh Sharma

Indian Institute of Technology Delhi

2016

Abstract

In today's growing era of social networking, e-commerce and online banking, security of the content being passed around over the internet has become more important than ever. To ensure unbreakable security and to prevent eavesdropping, it is essential to use authenticated, secured web services using SSL-based protocols like HTTPS. However, due to the inherent computationally intensive nature of cryptographic algorithms like RSA,AES,etc., the throughput of such secured web-services is relatively lower than that of unsecured services. In such a scenario, offloading the decryption of the encrypted data to a computationally superior device like a GPGPU may help in accelerating such secured web services, since the decryption algorithm can be executed in data-parallel, SIMD type of environments which the GPGPU certainly offers. We will also extend this approach to a heterogeneous computing environment consisting of multiple GPUs and CPUs on a single compute node. We also explore execution in a cluster of multiple such nodes using a message passing interface.

Acknowledgments

I express my sincere gratitude towards my guide **Prof. Subodh Sharma** for his constant help, encouragement and inspiration throughout the project work. Without his invaluable guidance, this work would not have been possible. I would also like to thank the IITD **HPC** group for providing a state-of-the-art platform which helped in performing various experiments during the project.

Gaurav Gupta
IIT Delhi

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In this chapter, we discuss about the problem definition with a quick background of *SSL* and the most popular public key cryptography technique - *RSA*. We then give an outline of this thesis.

1.1 Public Key Cryptography

In the traditional symmetric key cryptography, only one key is used which is shared by the sender and receiver. Various ciphers are available for encrypting the messages into ciphertext like AES, DES,etc. However, since the key is identical on both sides, it is vulnerable to attacks if the intruder can somehow gain access to the shared key. Public key cryptography on the other hand uses two keys - *public & private*. The receiver of the message has both the keys while the sender knows only about the public key shared by the receiver. The sender then encrypts the message with the public key and sends the ciphertext across to the receiver which is then decrypted by the receiver using the private key. Such systems use cryptographic algorithms which are based on mathematical problems which do not have polynomial time solutions. For example, the RSA is based on the fact that there is currently no polynomial time solution to integer factorization of large integers or integer factorization is an *NP* problem.

1.1.1 RSA

RSA is one of the most popular public key cryptographic systems widely being used today. It was named after its inventors - *Ron Rivest, Adi Shamir, and Leonard Adleman.* As stated earlier, it is based on the difficulty of factoring large numbers. It consists of a private key d and a public key pair *(e,n)*. The key generation process is explained below.

1. Two large different prime numbers *pandq* are chosen. For $k-bit$ encryption, *pandq* are $k/2$ bits each.
2. Calculate modulus $n = p.q$ where n is a k-bit integer.
3. Compute $\phi(n) = \phi(p)\phi(q) = (p-1).(q-1)$ where $\phi(n)$ is the Euler's totient function which counts the number of integers less than n which are co-prime to n.
4. Select e such that e and $\phi(n)$ are co-prime.
5. Calculate d such that $d.e \equiv 1(mod\phi(n))$.

Any message m can be encrypted using the public key (e, n) using the modular exponentiation,

$$c = m^e \,\%\, n$$

And a ciphertext can be decrypted at the receiver end using the private key d as,

$$m = c^d \,\%\, n$$

1.2 SSL

Secured Sockets Layer(SSL) is a cryptographic protocol which provides security for data transmission over computer networks. It provides privacy and data integrity between two communicating computers over a computer network. It is most commonly used in *HTTPS* protocol for connection encryption. The Fig 1.1 shows a typical handshake between a client and a server trying to establish a secure connection over SSL.

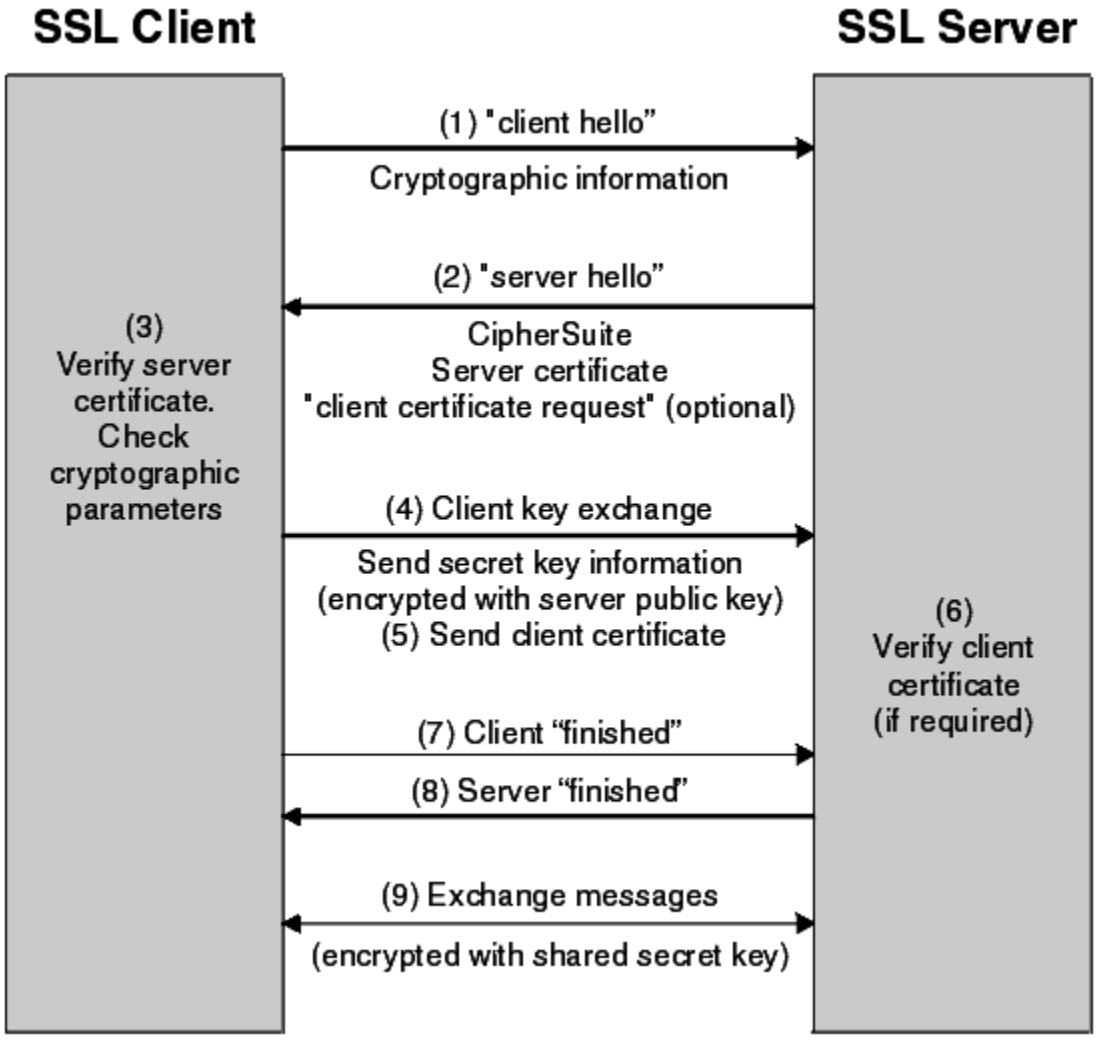


Figure 1.1: SSL Handshake[34]

In the *client hello* message, the client, typically a web browser, sends a list *cipher suites* supported by the client with some precedence order. A *cipher suite* contains

information about the key exchange algorithm like RSA, symmetric key encryption algorithm like AES, message authentication code (MAC) algorithm like SHA and a pseudoramdom function. The server responds with a *server hello* message containing the selected cipher suite that will be used in connection encryption and its digital certificate containing the public key. The client then verifies the digital certificate and sends an encrypted message containing the *pre-master secret* which will be used in the generation of symmetric shared keys on both sides. The encryption of the *pre-master secret* is typically done through a public-key cryptosystem like RSA using the server's public key shared in the digital certificate. The server needs to decrypt the encrypted *pre-master secret* using its private key. The client then sends a *finished* message encrypted with the shared key to which the server responds likewise to complete the handshake.

1.3 Problem Definition

In the SSL handshake, the decryption of the *pre-master secret* by the server is an expensive operation since it involves modular exponentiation($m = c^d \,\%\, n$) involving numbers which can be as large as 2048 bits. This results in low capacity of the server to handle more clients. To solve this problem, a number of *SSL accelerators* [19] are available in the market from leading vendors like *Cisco*. However, they are quite expensive and the cost per decryption may not be sustainable. Therefore, cheap commodity GPGPUs can be used in place of such accelerators to increase the server capacity. This essentially involves acceleration of RSA decryption by exploiting data parallelism in the decryption algorithm as well as request-level parallelism to increase the throughput. We discuss about this approach in the remainder of this thesis.

1.4 Thesis Outline

The thesis has been divided into the following chapters.

- Tools & Technologies - In this chapter, we discuss about the various tools and frameworks that were used during the development, verification and experimentation phases.
- Solution - Various algorithms with parallel implementations of operations and development iterations have been discussed in this chapter.
- Optimizations - This chapter details out all the optimizations used to increase the throughput in terms of *number of RSA decryptions per second*.
- Observations & Results - This chapter contains the numerical observations obtained through various experiments and the maximum observed throughput.
- Conclusion - In this chapter we discuss about the conclusion and contributions of this thesis.
- Future Work - An insight into the possible improvements and future work has been given in this chapter.

Chapter 2

Tools & Technologies

In this chapter, we discuss about various tools, frameworks and technologies which were used during the development, verification and experimentation phases.

2.1 General Purpose GPUs(GPGPUs)

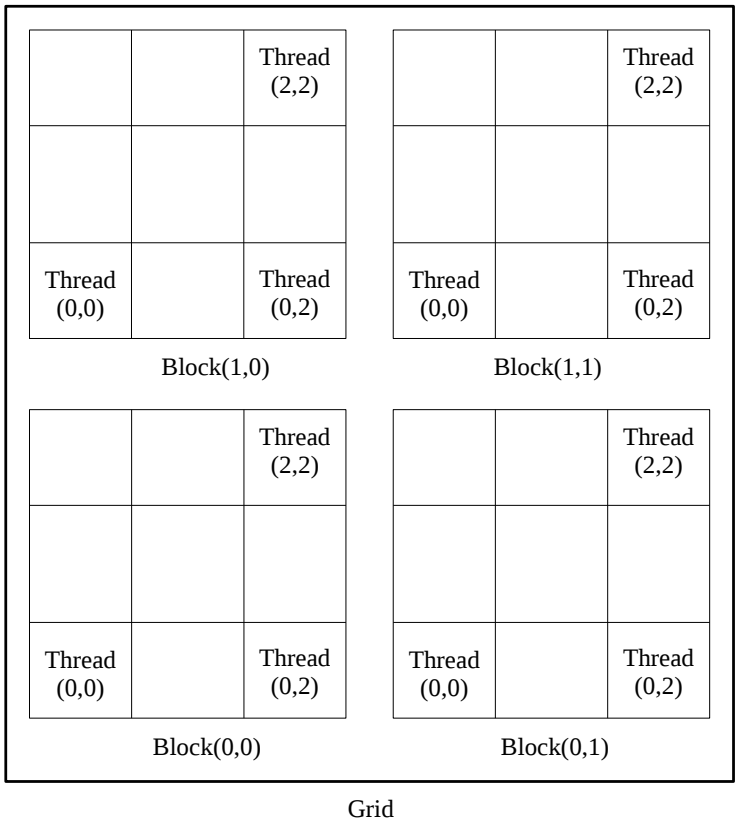
Graphics Processing Units or GPUs are devices which are essentially built on *SIMD* architectures. Unlike multicore CPUs which are MIMD, many-core GPGPUs contain a grid of *Streaming Processors(SP)*. Each SP contains a number of cores where one core takes care of one thread. At any given time, the same instruction is executed by all the cores in a SP unless the program has *divergence* in which case some cores sit idle. Such architectures are optimized for high multi-thread throughput and are ideal for execution of algorithms or programs which are highly data parallel. However, the memory bandwidth may become a bottleneck in such programs. Therefore, optimizations like *memory coalescing*, having enough number of threads per SP, etc. are essential to hide memory access latencies.

2.2 CUDA

The Compute Unified Device Architecture(CUDA) is a parallel computing platform and API model developed by *Nvidia*. It allows execution of programs on GPGPUs using the CUDA APIs. Code on the device is executed as a *kernel* which is essentially a C function. Kernels are invoked with additional parameters like *grid dimensions, shared memory size and a stream identifier* some of which are optional. Instructions are executed in a SP in a lock-step manner, i.e. all threads in a *warp* execute the same instruction. A *warp* is a group of threads(with typical size of 32) which is scheduled by the SP to execute at a time.

2.2.1 Thread Hierarchy

Each kernel is executed on a *grid* of *blocks*. A *block* is a collection of *threads* and is mapped to a single SP. Each *thread* executes on one core of the SP. The fig 2.1 shows the thread hierarchy pictorially.



A two dimensional grid of 4 blocks where each block is also two dimensional made up of 9 threads.

Figure 2.1: CUDA Thread hierarchy

2.2.2 Memory Hierarchy

Various kinds of memories exist depending upon the proximity and program scope. Selection of memory for storage and retrieval of data is a crucial decision which can greatly affect the performance of the program. The types of available memories with location and scope is given below an in fig 2.2.

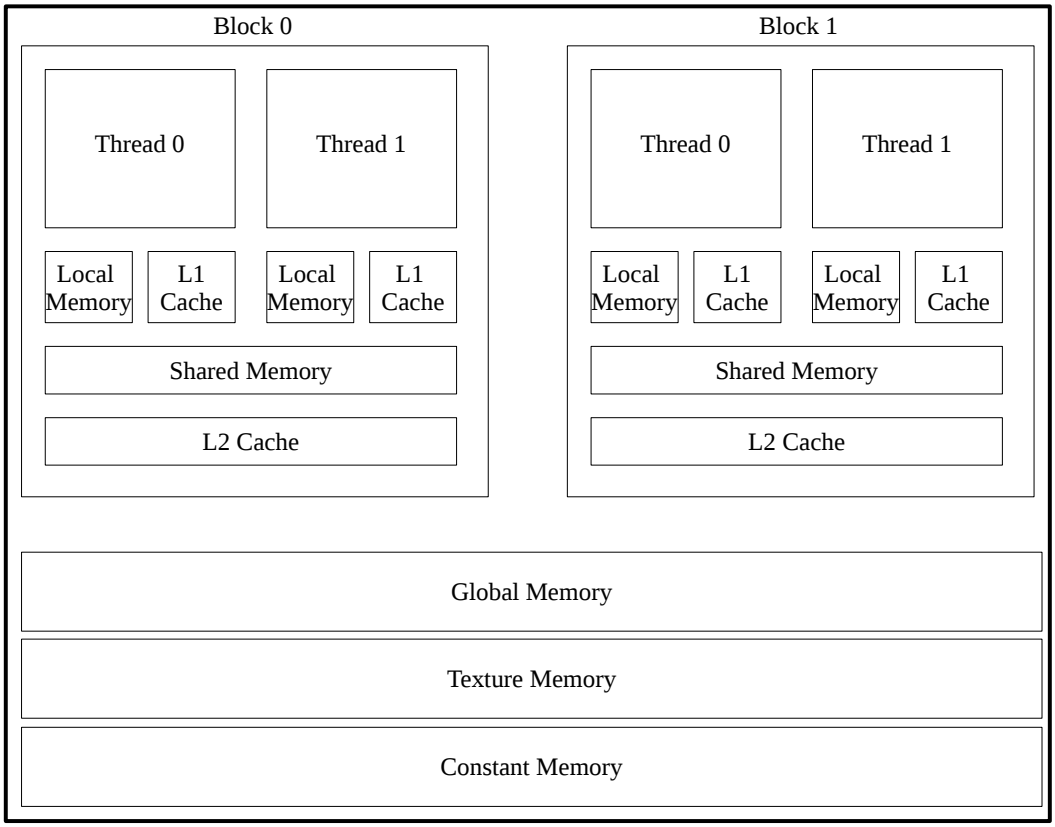


Figure 2.2: CUDA Thread hierarchy

- *Global memory* – off-chip, accessible to all blocks in a grid.
- *Local memory* – off-chip, accessible only to a thread in a block.
- *Registers* – accessible only to a thread in a block.
- *Shared memory* – on-chip, accessible only to threads in a block.
- *L1 cache* – part of shared memory, accessible only to threads in a block.
- *Texture and Constant memory* – read-only memories, accessible to all blocks in a grid.

2.3 GKLEE

To ensure correct program behaviour and prevent unexpected run-time crashes, verification of the code becomes an essential part, especially, in the context of parallel programs.

GKLEE [36] is a verifier-cum-analyzer for CUDA programs which detects various defects like invalid memory accesses, read-write & write-write data races among threads, shared memory bank conflicts, etc. Examples of such defects uncovered by GKLEE are given below.

1. *Data Races*: Data races occur when multiple threads try to access the same memory location and atleast one of them is performing a write operation to that location. This results in an unexpected program behaviour and leads to different outputs for the same inputs. Such races can either occur among threads in the same warp or different warps. For example, in the kernel below, there is a write-write race between a thread with id 0 executing the second iteration(value of variable $ind = 1$) and a thread with id 1 executing the first iteration(value of variable $ind = 1$). Similarly a read-write data race also exist among threads of different warps.

```
__global__ void pmul(...)
{
        int x = threadIdx.x;
        for(i=0;i<k;i++){
                product = m[i]*ndig;
                radix_type lword = product%2;
                radix_type hword = product/2;
                int ind = x+i;

                radix_type carry =(inter_buf[ind]+lword)
                   /2;
                inter_buf[ind] = (inter_buf[ind]+lword)
                   %2;
                carry_buf[ind+1] += carry;

                carry = (inter_buf[ind+1]+hword)/2;
                inter_buf[ind+1] = (inter_buf[ind+1]+
                   hword)%2;
                carry_buf[ind+2] += carry;

        }
}
```

The above problem is duly detected by GKLEE as shown below.

```
********* Start checking races at Device Memory
   **********

[GKLEE]: Under the pure canonical schedule, within a
   block,
thread 0 and 1 incur a Write-Write race with the same
   value (Benign) on
[GKLEE] Inst:
Instruction Line: 39, In File: bkernels.cpp, With Dir
   Path: /
[File:bkernels.cpp, Line: 39, Inst:
```

```
inter_buf[ind+1] = (inter_buf[ind+1]+hword)&2;]
  store i8 %98, i8* %103, align 1, !dbg !1038
<W: 31930816, 1:0, b0, t0>
[GKLEE] Inst:
Instruction Line: 34, In File: bkernels.cpp, With Dir
   Path: /
[File: bkernels.cpp, Line: 34, Inst:
inter_buf[ind] = (inter_buf[ind]+lword)&2;]
  store i8 %59, i8* %63, align 1, !dbg !1035
<W: 31930816, 1:0, b0, t1>
```

The solution to the above problem is to use a barrier synchronization(*__syncthreads*() in CUDA) so that threads in different warps are in the same loop iteration. GKLEE also reports shared memory *bank conflicts* which occur when threads in a warp are accessing words belonging to the same shared memory bank. This results in serialization of requests to the same bank and incurs a time penalty which may be significant. Therefore, the data access patterns of different threads becomes an important aspect.

2. *Invalid Memory Access*: Invalid memory accesses to device memory can cause unexpected program crashes. GKLEE detects such invalid accesses and reports them so that they can be handled correctly. For example in the following kernel, the variable *ind* is of type *char*. Hence, this can lead to an overflow if value of $x + i$ is greater than 127 resulting in a negative value.

```
typedef char radix_type;
__global__ void pmul(...)
{
        ......
        for(int i=0;i<k;i++){
                ...
                radix_type ind = x+i; //-128<=ind<128
                radix_type carry = (inter_buf[ind]+lword
                   )/2;
                inter_buf[ind] = (inter_buf[ind]+lword)
                   %2;
                carry_buf[ind+1] += carry;

                 __syncthreads();

                carry = (inter_buf[ind+1]+hword)/2;
                inter_buf[ind+1] = (inter_buf[ind+1]+
                   hword)%2;
                carry_buf[ind+2] += carry;

        }
}
```

GKLEE highlights this problem as shown below.

```
type:2  flow:0  instr:  %43 = load i8* %42, align 1, !
   dbg !1034  cond: merged:
```

```
KLEE: ERROR: bkernels.cpp:33: memory error: out of bound
    pointer
```

2.4 OpenMP

Open Multi-Processing or OpenMP [38] is an API to perform multi-processing on a shared-memory platform. A number of directives are available for parallelizing a section of code to execute in a *Single Program Multiple Data*(SPMD) fashion on a set of available processors. The fig 2.3 shows the execution of a program with sections of code executed parallely using OpenMP.

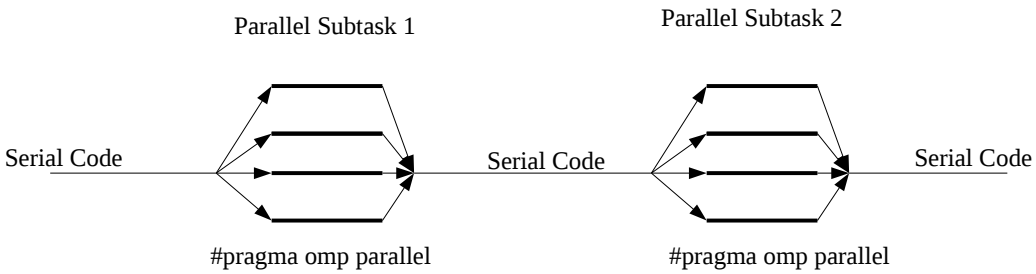


Figure 2.3: Execution with OpenMP

2.5 Open MPI

Open MPI [37] is a high performance Message Passing Interface which enables efficient usage of the available hardware. In is used in building large distributed highly scalable parallel systems. It also allows efficient communication between nodes of different *ranks*. On high performance computing platforms consisting of a large number of computing nodes, it acts as a horizontally scalable *middleware* for spawning the same process on different nodes (*SPMD*) and inter-node communication to work together towards a global solution.

Chapter 3

Solution

In this chapter, we discuss about various algorithms and techniques which together constitute a solution to the problem at hand. Various revisions to different algorithms and development iterations have also been explained to give an insight into how the solution evolved over time. Later, in the next chapter, a detailed analysis of the effects of using different techniques and subtleties has been worked upon.

3.1 Number Representation

As discussed in section rsa_public_crypt, for a k-bit RSA decryption, the maximum value of any of the k-bit RSA parameters - n,e & d can theoretically be $2^{k-1}-1$. Now, k can be as large as 4096 bits which cannot be stored in the regular data types like integers or long integers available in most of the modern programming languages. Hence, it is necessary to use a multiprecision number system with a relevant base and an appropriate data structure to store and process such big numbers. Any value N can be represented in a base system b as

$$N = N_{k-1}b^{k-1} + N_{k-2}b^{k-2} + ... + N_1b^1 + N_0$$

Therefore, we only need to store the coefficients $N_{k-1}, N_{k-2}, ..., N_0$ in a suitable data structure for storage and processing. To reduce the number of coefficients i.e. to minimize k, it is apparent that the base b should be as large as possible. Since, $0 \leq N_i < b$, and the size of an integer is typically 32 bits(or 4 bytes), a decent choice for the base b could be 2^{16} or 2^{32}. We chose the base as 2^{32} so that arithmetic overflows can be implicitly handled by long integers(64 bits) and explicit caution for carry handling can be avoided. Note that the RSA parameters obtained from a higher application layer may not always be in the desired base. In such cases fast subroutines to convert bit strings or strings of decimal digits into a specified base b are available which can be use to pre-process and normalize the inputs before passing them to the main procedure.

3.2 Data Structure

The most intuitive data structure to store the coefficients $N_{k-1}, N_{k-2}, ..., N_0$ would be a simple array of integers as shown below. Such data structure helps simplifying parallel algorithms using a framework like CUDA and also improves the efficiency of the program by exhibiting spatial locality.

```
#define BITLENGTH 16
#define BASE 32
typedef long radix_type;
radix_type n[BITLENGTH];
```

Figure 3.1: Data Structure to represent RSA parameters

3.3 Montgomery Multiplication Algorithm

The RSA decryption equation can be rewritten using the following identities of modular arithmetic,

$$a.b \,\%\, m = (a \,\%\, m).(b \,\%\, m) \,\%\, m$$

as

$$m = c^d \,\%\, n = (c.c...d\,times) \,\%\, n = ((((c.c \,\%\, n).c) \,\%\, n....).c \,\%\, n$$

Now since c *and* n are very large numbers, the modular multiplication sub-expression $(c.c \ \% \ n)$ becomes inherently expensive due to trial division by such a large modulus n. Therefore, we need an algorithm which could get rid of this trial division by a big number. Montgomery multiplication [3] is such an algorithm which avoids such division by using an appropriate number R. Generally, for a k-bit modular multiplication, R is chosen to be 2^k such that R *&* n are coprime and $R > n$. Using such R, multiplication and division by R become bitwise left and right shifts respectively which are very efficient operations from a hardware perspective as well. In addition the modulus operation becomes a bit masking operation. In order to calculate a modular multiplication such as $m = c.c \,\%\, n$, we first need to convert each of the operands into a montgomerized form using the chosen value of R as shown below.

$$\overline{c} = c.R \,\%\, n$$

where $R = 2^k$
The Montgomery multiplication algorithm is shown below.

Algorithm 1 Montgomery Multiplication Algorithm

Input: Montgomerized form of a *&* b - $\overline{a}\&\overline{b}$
R^{-1} where R.$R^{-1} \equiv 1$ (mod m)
m' such that R.$R^{-1} - m.m' = 1$
Output:$\overline{a}.\overline{b}.R^{-1} \,\%\, m$
$T = \overline{a}.\overline{b}$
$M = T.m' \,\%\, R$
$U = (T + M.m)/R$
  if $U \geq m$ **then**
    return $U - m$
  else
    return U
  end if

Let us call the above algorithm as $montgomery(a, b, m)$ which calculates $a.b.R^{-1} \,\%\, m$. In addition to the operands $a, b\&m$, the algorithm also requires R^{-1}, m' which can

be pre-computed using available known algorithms and passed as inputs to the function.Also notice that the algorithm requires montgomerized form of the input operands. This again faces the problem of expensive division by modulus m. However, we can calculate the montgomerized form of the operands using the algorithm with R^2 as one of the operands as shown below.

$$\overline{c} = c.R \,\%\, n = c.R^2.R^{-1} \,\%\, n = montgomery(c, R^2, m)$$

Also note that the result of the algorithm needs to be converted from montgomerized form to the normal form which can be easily done as shown below.

$$c = \overline{c}.1.R^{-1} \,\%\, n = montgomery(\overline{c}, 1, m)$$

3.4 Parallel Operations

The algorithm given in section mont_section is sequential and we want to induce some form of parallelism by using parallel versions of addition, multiplication, subtraction and shift operations on large numbers. We discuss about the implementation of these operations as CUDA kernels in the next few sub-sections and a revised Montgomery multiplication algorithm which uses these parallel operations.

3.4.1 Parallel Multiplication

A number of algorithms exist for multiplication of big numbers. A sequential algorithm has an overall time complexity of $O(n^2)$ and is slower for large input size. We discuss a couple of parallel algorithms with complexities $O(nlogn)$ and $O(n)$ respectively.

Algorithm 1

This algorithm has two phases. In phase 1, each digit of one operand is calculated with all digits of the second operand to form *partial products*. One thread calculate one partial product. Hence, the number of threads used is equal to the number of digits in one operand. In phase 2, all partial products are reduced and merged into bigger partial products by shifting and adding them together as shown in the fig 3.3. The time complexities of the two phases are $O(nlogn)$ and $O(n)$ respectively which leads to an overall complexity of $O(nlogn)$.

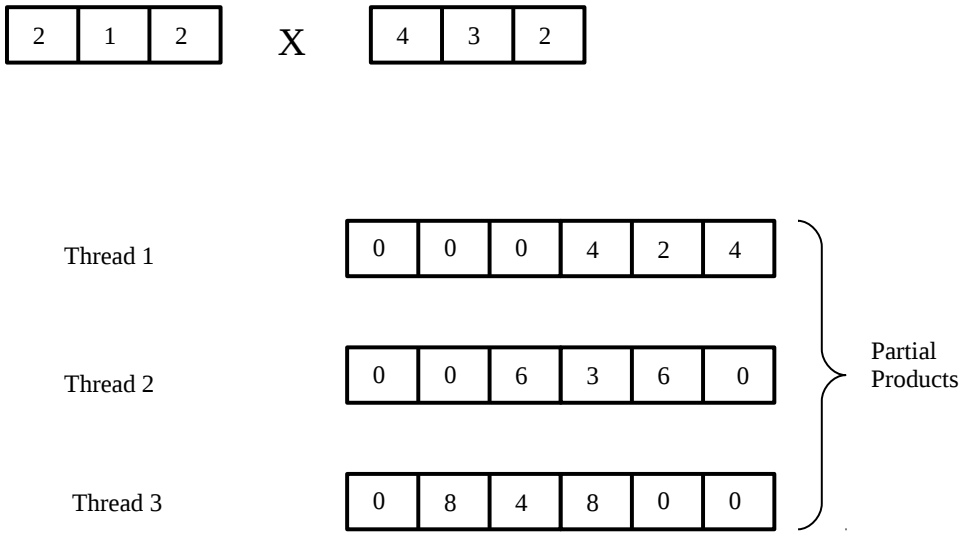


Figure 3.2: Parallel Multiplication Algorithm 1 Phase 1

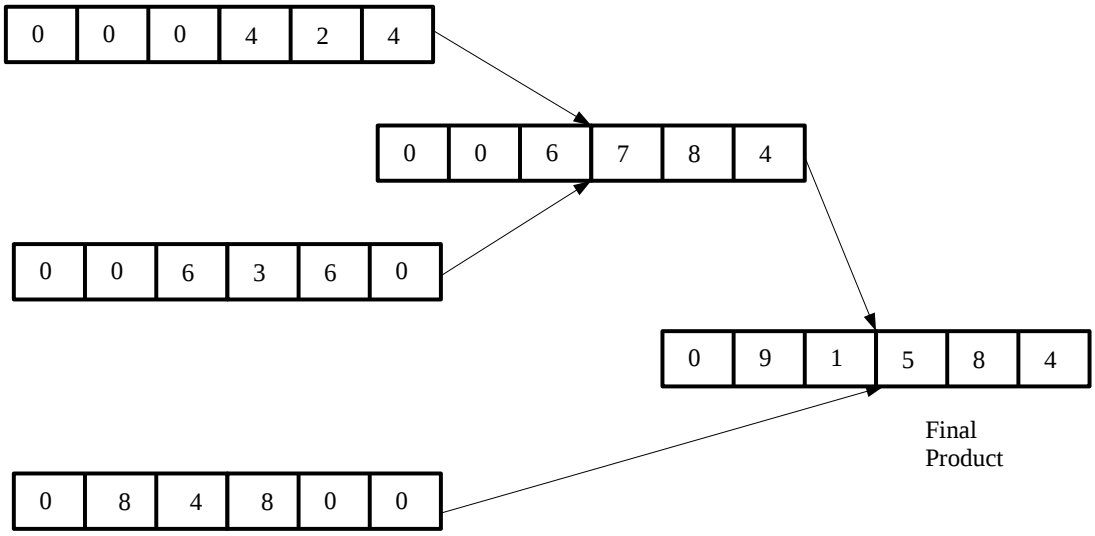


Figure 3.3: Parallel Multiplication Algorithm 1 Phase 2

```
//Phase 1 - Partial Products Formation
__global__ void parallel_mul(radix_type *m, radix_type *n,
    int k,radix_type *res, int stridem,int striden){
        extern __shared__ radix_type partial_prod[];//size:k
            *k*2
        int bx = blockIdx.x;
        int x = threadIdx.x;
        int i;
        long_radix_type ndig = n[bx*striden+x];
        long_radix_type product = 1;
        short pos = 2*k*x+x;
        for(i=0;i<k;i++){
                product = m[bx*stridem+i]*ndig;
                partial_prod[pos+i] = (partial_prod[pos+i]+
                    product) & BASEMINUS1;
                partial_prod[pos+i+1] = (partial_prod[pos+i
                    ]+product) >> BASE;
        }
        reduce(partial_prod,k,x);
        if(x==0){
                for(int i=0;i<2*k;i++){
                        res[2*k*bx+i] = partial_prod[i];
```

```
                }
        }

}

//Phase 2 - Partial Products Reduction
__device__ void reduce(radix_type *partials, int k, int tx){

        int next_pow2 = 1;
        int res_size = 2*k;
        while((k-1)>=next_pow2){
                if((tx%(next_pow2*2)==0) && (tx+next_pow2)
                   <=(k-1)){
                        acc(&partials[(tx)*(res_size)],&
                           partials[(tx+next_pow2)*(res_size
                           )],res_size);
                }
                next_pow2 *=2;
        }
}
```

Algorithm 2

A better algorithm for parallel multiplication with $O(n)$ was proposed in [1]. In this algorithm also, each thread takes care of one digit of one of the operands. It then multiplies that digit with every digit of the other operand, calculates low & high words and appropriately adds the partial products to the intermediate and carry buffers which are shared among the threads in a shared memory. In the end, the carry buffer is added to the intermediate buffer to obtain the final product.

```
__global__ void pmul(radix_type *mg, radix_type *n, int k,
   radix_type *res,
                int stridem, int striden, int num_threads) {
        extern __shared__ radix_type shared_mem[]; //size
           :(2*k+2*k+k)*(num of req per block)
        int bx = blockIdx.x;
        int tx = threadIdx.x;

        if (tx < num_threads) {
                int reqs = num_threads / k;
                int reqno = tx / k;
                radix_type *carry_buf = &shared_mem[0 +
                   reqno * 2 * k]; //size:k*2*(num of req
                   per block)
                radix_type *inter_buf = &shared_mem[(reqs) *
                    2 * k + reqno * 2 * k]; //size:k*2*(num
                   of req per block)
                radix_type *m = &shared_mem[(reqs) * 4 * k +
                    reqno * k]; //size:k*(num of req per
                   block)
```

```
int i;
int x = tx % k;
m[x] = mg[bx * stridem * reqs + stridem *
   reqno + x];
carry_buf[x] = 0;
carry_buf[x + k] = 0;
inter_buf[x] = 0;
inter_buf[x + k] = 0;

//digit of one operand that will be taken
   care of by this thread..
long_radix_type ndig = 1;
ndig = n[bx * striden * reqs + striden *
   reqno + x];

long_radix_type product = 1;

for (i = 0; i < k; i++) {
        product = m[i] * ndig;
        //low word
        radix_type lword = product &
           BASEMINUS1;
        //high word
        radix_type hword = product >> BASE;
        //store and add to partial results
           with carry handling
        int ind = x + i;
        //lword
        radix_type carry = (inter_buf[ind] +
            lword) >> BASE;
        inter_buf[ind] = (inter_buf[ind] +
           lword) & BASEMINUS1;

        __syncthreads(); //only one barrier
           required only if the
           THREADSPERBLOCK>WARP SIZE

        //hword
        if (ind + 1 < 2 * k) {
                carry_buf[(ind + 1)] +=
                   carry;
                carry = ((inter_buf[ind + 1]
                    + hword) >> BASE);
                inter_buf[ind + 1] = (
                   inter_buf[ind + 1] +
                   hword) & BASEMINUS1;
        }
        if (ind + 2 < 2 * k) {
```

```
                                carry_buf[(ind + 2)] +=
                                    carry;
                            }

                    }
                    if (x == 0) {
                            //Add carry buffer to intermediate
                                buffer
                            acc(&inter_buf[0], &carry_buf[0], 2
                                * k);
                    }
            }
}
```

3.4.2 Parallel Addition

An $O(n)$ parallel addition algorithm is shown in figure 3.4. Each thread calculates the sum of digits at the same position, generates carry to the shared carry buffer. In the worst case, a carry generated at the least significant position causes subsequent carry generations upto the most significant positions. To handle this case, each thread must run through a loop with number of iterations equal to the bit length.

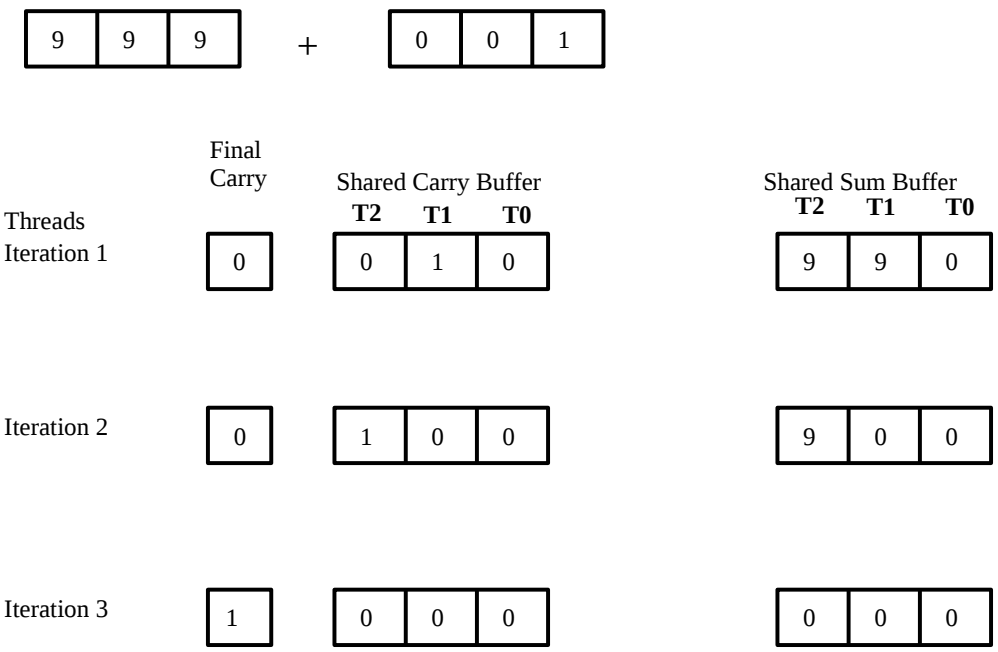


Figure 3.4: Parallel Addition Algorithm

```
__global__ void padd(radix_type *a, radix_type *b, int k,
   int num_req,
                int stridea, int strideb, radix_type *
                   residue_carry, int num_threads) {
        extern __shared__ radix_type shared_mem[];      //
           size:k+k+1

        int tx = threadIdx.x;
        int bx = blockIdx.x;
        int reqs = num_threads / k;
        int reqno = tx / k;

        radix_type *res = &shared_mem[0 + reqno * k];   //
           size:k
        radix_type *carry_buf = &shared_mem[reqs * k + reqno
            * (k + 1)];         //size:k

        int x = tx % k;
        carry_buf[x] = 0;
        res[x] = 0;
        if (x == 0) {
                carry_buf[k] = 0;
        }

        long_radix_type sum = 0;

        sum = a[bx * reqs * stridea + reqno * stridea + x]
                                + b[bx * reqs * strideb +
                                   reqno * strideb + x];
        for (int i = 0; i < k; i++) {
                res[x] = sum & BASEMINUS1;
                carry_buf[x + 1] = carry_buf[x + 1] + (sum
                   >> BASE);
                sum = res[x] + carry_buf[x];
                carry_buf[x] = 0;
        }
        if (x == k - 1) {
                residue_carry[bx * reqs + reqno] = carry_buf
                   [k];
        }
        a[bx * reqs * stridea + reqno * stridea + x] = res[x
           ];
}
```

3.4.3 Parallel Subtraction

An $O(n)$ parallel subtraction algorithm has been used. Each thread calculates the difference of digits at the same position, generates borrow to the shared borrow buffer. As in the case of parallel addition, in the worst case, a borrow generated at the least significant

position causes subsequent borrow generations upto the most significant positions. To handle this case, each thread must run through a loop with number of iterations equal to the bit length.

```
__global__ void psub(radix_type *a, radix_type *b, int k,
    radix_type *resg,
                radix_type *add_carry, int stridea, int
                    strideb, int compare,
                int num_threads) {
        extern __shared__ radix_type shared_mem[];      //
            size:k+k

        int bx = blockIdx.x;
        int tx = threadIdx.x;
        int reqs = num_threads / k;
        int reqno = tx / k;
        radix_type *res = &shared_mem[0 + reqno * k];   //
            size:k
        radix_type *carry_buf = &shared_mem[(reqs + reqno) *
             k];         //size:k+1
        int x = tx % k;

        int pos = bx * reqs + reqno;
        if ((compare == 1
                        && (d_compare(&a[stridea * pos], &b[
                            strideb * pos], k) >= 0))
                        || !compare || (add_carry[pos] == 1)
                            ) {
                carry_buf[x] = 0;
                res[x] = 0;
                radix_type carry = 0;
                int dig = 0;
                dig = a[stridea * pos + x] - b[strideb * pos
                    + x];
                for (int i = 0; i < k; i++) {
                        carry = dig < 0;
                        res[x] = ((carry) * (BASEMINUS1 + 1)
                            + dig);
                        if ((x + 1) != k)
                                carry_buf[x + 1] = carry;
                        dig = res[x] - carry_buf[x];
                }
                resg[k * pos + x] = res[x];
        }
}
```

3.4.4 Parallel Right Shift

One of the major advantages of using the Montgomery multiplication algorithm is that a division by R where $R = 2^k$ becomes a right shift operation which can be parallelized

as shown below. Each thread copies one digit or coefficient from position $i + shifts$ to i as shown in the fig 3.5.

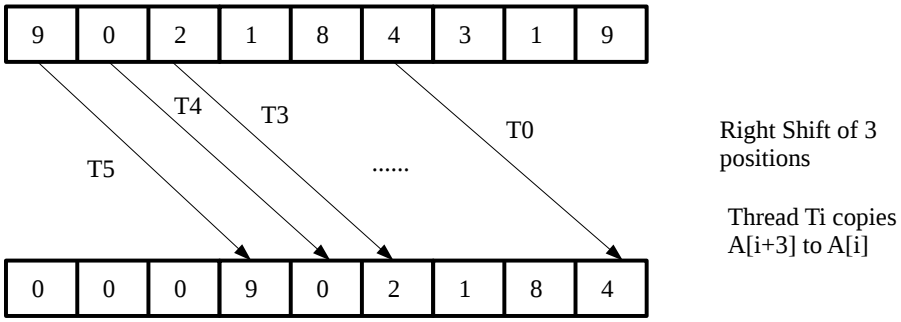


Figure 3.5: Parallel Right Shift Algorithm(Division by R)

```
__global__ void right_shift(radix_type *m, int stride, int k
    , radix_type *res,
                int num_threads) {
        int bx = blockIdx.x;
        int tx = threadIdx.x;
        int reqs = num_threads / k;
        int reqno = tx / k;
        int x = tx % k;
        int pos = bx * reqs + reqno;

        res[pos * stride + x] = m[pos * stride + x + k];
}
\FloatBarrier
```

3.4.5 Revised Montgomery Multiplication

Let us denote the parallel versions of arithmetic operations multiplication, addition, subtraction and shift as *parMul, parAdd, parSub & parShift*. Then the revised parallel Montgomery modular multiplication *parMontgomery* is given below.

Algorithm 2 Parallel Montgomery Multiplication Algorithm

Input: Montgomerized form of a & b - $\overline{a}$&$\overline{b}$
R^{-1} where R.$R^{-1} \equiv 1$ (mod m)
m' such that R.$R^{-1} - m.m' = 1$
Output:$\overline{a}.\overline{b}.R^{-1} \,\%\, m$
$T = parMul(\overline{a}, \overline{b})$
$M = parMul(T, m') \,\%\, R$
$U = parShift(parAdd(T, parMul(M, m)), k)$
  if $U \geq m$ **then**
    return $parSub(U, m)$
  else
    return U
  end if

3.4.6 Square and Multiply Exponentiation Algorithm

We know that in RSA, to obtain the message m from ciphertext c we need to calculate the modular exponentiation equation $m = c^d \,\%\, n$. Since d can be as large as 4096 bits, it is practically infeasible to compute this modular exponentiation using a naive approach. To reduce the number of exponentiations, we use the Square & Multiply algorithm. This algorithm is based on the recursive definition to calculate x^n as shown below.

$$x^n = \begin{cases} x.(x^2)^{(n-1)/2} & \text{if } n \text{ is odd} \\ (x^2)^{n/2} & \text{if } n \text{ is even} \end{cases}$$

To calculate the modular exponentiation using the Square and Multiply algorithm, we first convert the exponent d into binary notation and then use the following algorithm. Note that, int this algorithm, the modular multiplication is carried out using the parallel Montgomemary multiplication algorithm as described in the previous sections.

Algorithm 3 Modular Exponentiation Using Square And Multiply

Input: c, d&n
Output:$m = c^d \,\%\, n$
$m = 1$
  for $i = BITLENGTH - 1$ **to** 0 **do**
    $m = parMontgomery(m, m, n)$ # $m = m^2 \,\%\, n$

    if $d_i = 1$ **then**
      $m = parMontgomery(m, c, n)$ # $m = m.c \,\%\, n$

    end if
  end for

3.5 Implementation Iterations

3.5.1 Iteration 1

The first version of the implementation revolved largely around the use of CUDA's *dynamic parallelism* [31] which allows a kernel to be called from within another kernel.

We wanted that all the instructions should be executed by the GPU only with minimum interference with the CPU's work. However, as discussed in [7], dynamic parallelism suffers from non-trivial runtime overheads and parent-child communication takes place through the off-chip global memory which incurs a lot of memory clock cycles. In addition to this, only one sub-kernel can be active at any given time thereby serializing the execution of multiple sub-kernel invocations by multiple threads in a block. The Fig 3.6 shows the nested kernel call tree of depth 4. For a single 1024-bit decryption, this implementation took around 3.5 seconds which is not acceptable. For latency-critical applications, dynamic parallelism may not be the right solution, hence we changed our implementation to remove nested kernel calls and use collaboration of CPU and GPU to perform the decryption.

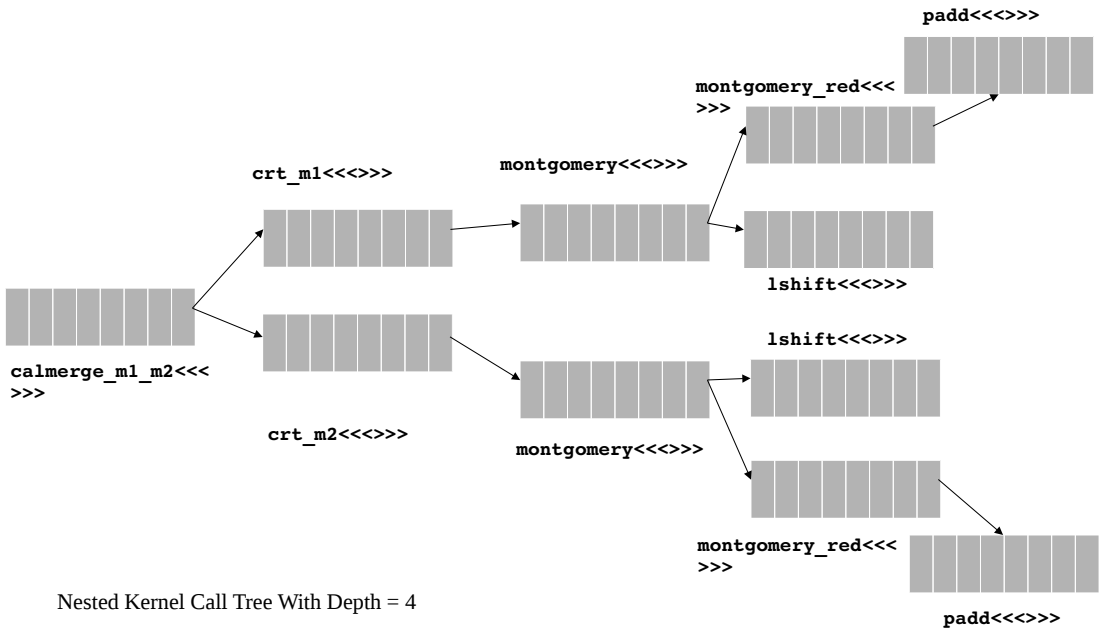


Figure 3.6: Nested kernel call tree

3.5.2 Iteration 2

As discussed in the previous section, dynamic parallelism induces non-trivial runtime overheads. We change our implementation to use the collaboration of CPU and GPU rather than GPU doing all the work. The serial portions of the program are executed by the CPU while the code fragments which can be parallelized are offloaded to the GPU. For example, in the Montgomery Multiplication algorithm discussed before, parallel operations like *parMul, parAdd, etc.* are essentially kernel calls while in the modular exponentiation algorithm, the *for* loop and *if* condition are executed by the CPU. This also presents an opportunity to utilize the idle waiting time spent by CPU in between

the kernel invocation and device synchronization runtime calls to execute some other task or process some of the requests while GPU process other requests. This idea is used in the later implementations along with request batching.

3.5.3 Iteration 3

GPGPUs are throughput architectures [15]. Their power lies in the way they operate in *SIMD* fashion. In addition to discovering data parallelism in algorithms, it is essential to use request level parallelism as well to unleash the full potential of a graphics card. To keep the GPU busy we should design the CUDA kernels such that *Occupancy* [21] is as high as possible. This brings factors like *grid dimensions, shared memory per block, number of registers per block, number of active blocks per SM* into the equation which should be carefully taken into consideration while invoking the kernels. For a good occupancy and multiprocessor activity we should enough number of active warps that can be scheduled in any active cycle in each SM. Thus we should ensure that the block dimensions provide enough work to a SM. We calculate the *optimal runtime grid parameters* using a empirical approach which is discussed in the optimizations section(Optimization 5).In addition to the occupancy, we have used apropriate striding to ensure that the global memory accesses generated from a block are coalesced. This is again discussed in detail in the optimizations section.

3.5.4 Iteration 4

We want to build a solution which is scalable both *vertically and horizontally* [4]. We use *OpenMPI* [37] library to achieve horizontal scalability by executing the program in *SPMD* fashion on multiple available nodes, each having a number of GPUs and CPU cores. Vertical scalability is achieved by spawning multiple CPU threads on the same node and assigning one CUDA *Stream* to each thread so that each host thread can independently invoke kernels concurrently with other host threads.

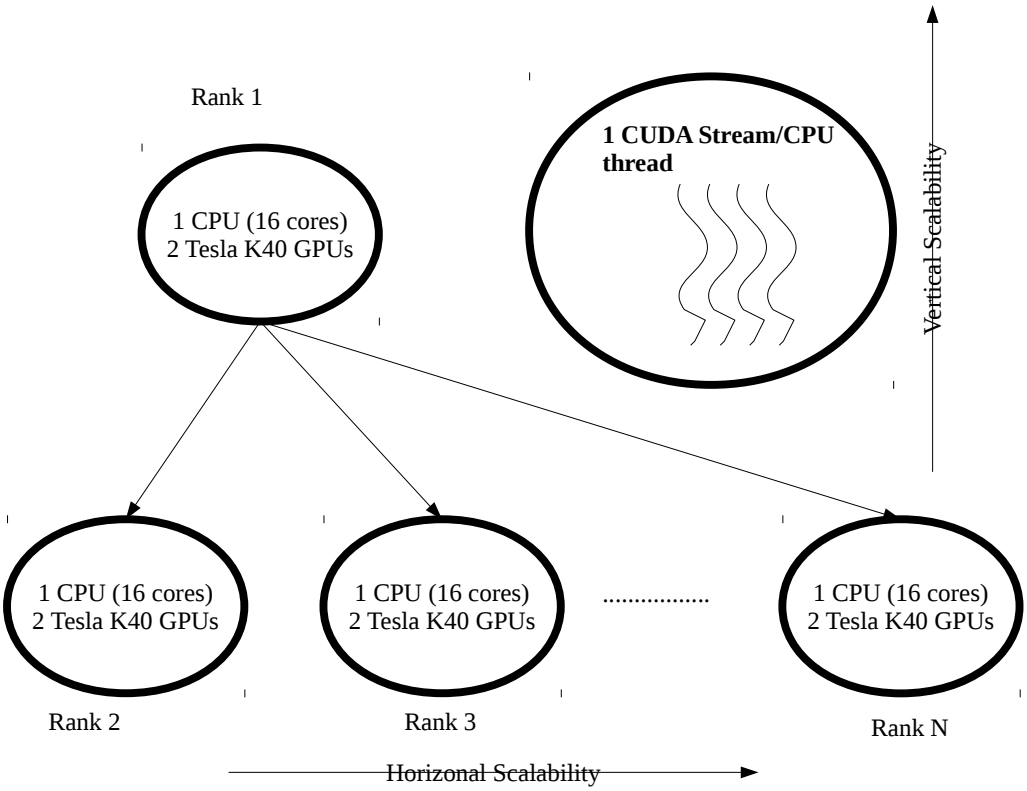


Figure 3.7: Vertical and Horizontal Scalability

3.5.5 Iteration 5

As pointed out before, we would like to utilize the idle waiting time of the CPU between kernel call and device synchronization call. We achieve this by dividing the request operations between CPU and GPU so that both have some task to work upon at any given time. Since the multiplication and addition of large integers are the most expensive operations, we have created serial algorithms for multiplication and addition operations with complexities $O(n^2)$ and $O(n)$ respectively. However, since the parallel multiplication algorithm is $O(n)$, the serial multiplication algorithm is comparatively slower. Also, for execution on CPU side, the data needs to be transferred from GPU's global memory to host memory since the intermediate data on which operation like multiplication and addition are to be performed were allocated on the device. We use asynchronous data transfer using a separate CUDA stream so that the kernel execution and data transfer can overlap [23]. Please note that explicit data transfer is required when neither unified memory [10] nor host mapped pinned memory is used.

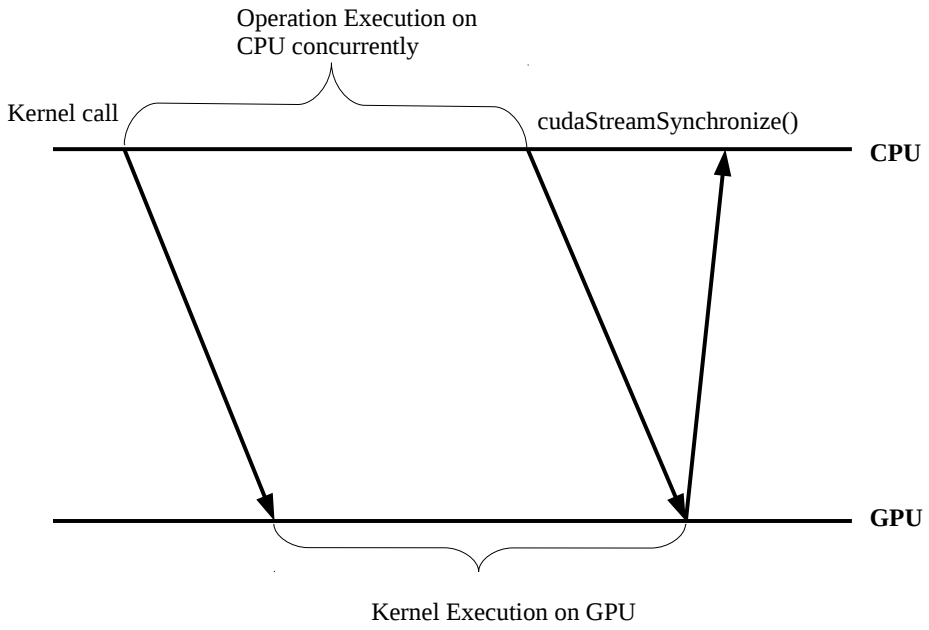


Figure 3.8: CPU-GPU task sharing

The main problem with the above approach is that the window available for execution on CPU is very small (few hundreds microseconds) since the kernels execution is quite fast. As a result, the serial CPU version should be able to finish in that time frame otherwise it may adversely impact the overall latency. One solution is to empirically determine the number of requests that can be processed by the sequential implementation in that time frame using test data. This gives us the number of requests that can be accommodated by serial versions of operations like addition, multiplication,etc. Now,

these sequential operations are invoked by the thread which invoked parallel operations via kernels earlier. We can also spawn separate available CPU threads to execute complete sequential versions of the algorithm. We have explored the following alternatives for the sequential execution of modular exponentiation in separate threads.

- A sequential version of the algorithms obtained by transforming CUDA kernels into serially executable code on CPU.
- Faster alternatives in other languages like Python - $pow(a, b, m)$ [35].
- Call external Python script from C code using pipes for inter-process communication.

The Fig 3.9 shows the heterogeneous model adopted in this implementation.

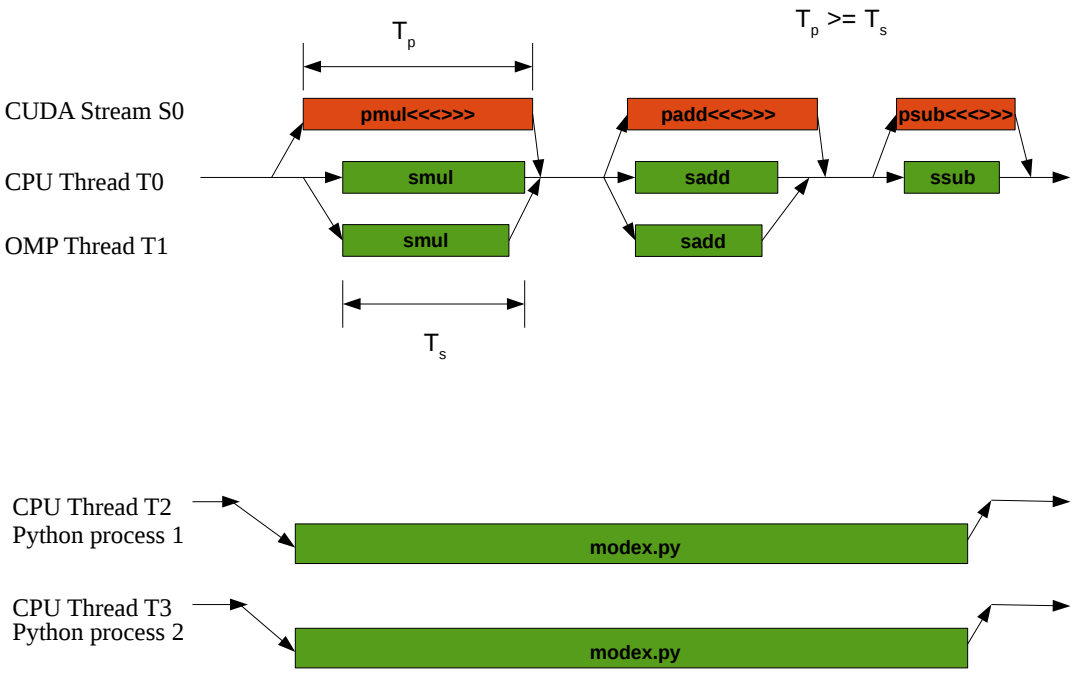


Figure 3.9: Heterogeneous Model

Chapter 4

Optimizations

In this chapter, we discuss about various optimizations[1][5][6][8][9][32][33] that were applied to decrease the latency per decryption and increase the throughput(*decryptions per second*). In the next chapter, we will discuss about the effect of these optimizations on latency and throughput numerically.

4.1 Optimization 1: Chinese Remainder Theorem

As we know that in RSA decryption, we are essentially evaluating a modular exponentiation equation, $m = c^d \,\%\, n$, where $c, d\&n$ are k-bit parameters. The Chinese Remainder Theorem [2] breaks down this one modular exponentiation equation into two independent modular exponentiations $m_1 = c^{d\,\%\,(p-1)} \,\%\, p$ and $m_2 = c^{d\,\%\,(q-1)} \,\%\, q$. The RSA parameters $p\&q$ being $k/2$ bits in length, help in reducing the overall time complexity the algorithm. In addition to this, the calculation of $m_1\&m_2$ can be done in parallel as they are independent of each other.

Now that we have two independent modular exponentiations, we need a way to calculate m from $m_1\&m_2$. The algorithm below calculates m from $m_1\&m_2$ using the formula given below.

$$m = m_2 + [(m_1 - m_2).q^{-1} \,\%\, p].q$$

Algorithm 4 Calculation of m using $m_1\&m_2$

Input: $m_1 = c^{d\,\%\,(p-1)} \,\%\, p$
$m_2 = c^{d\,\%\,(q-1)} \,\%\, q$
p, q, q^{-1}
Output:m
$X = parSub(m_1, m_2)$
$Y = parMontgomery(X, q^{-1}, p)$
$Z = parMul(Y, q)$
$m = parAdd(m_2, Z)$

4.2 Optimization 2: Memory Pooling

The CUDA runtime API for memory allocation, management and deallocation includes method calls such as *cudaMalloc, cudaMemcpy and cudaFree*. Such calls are *blocking* in

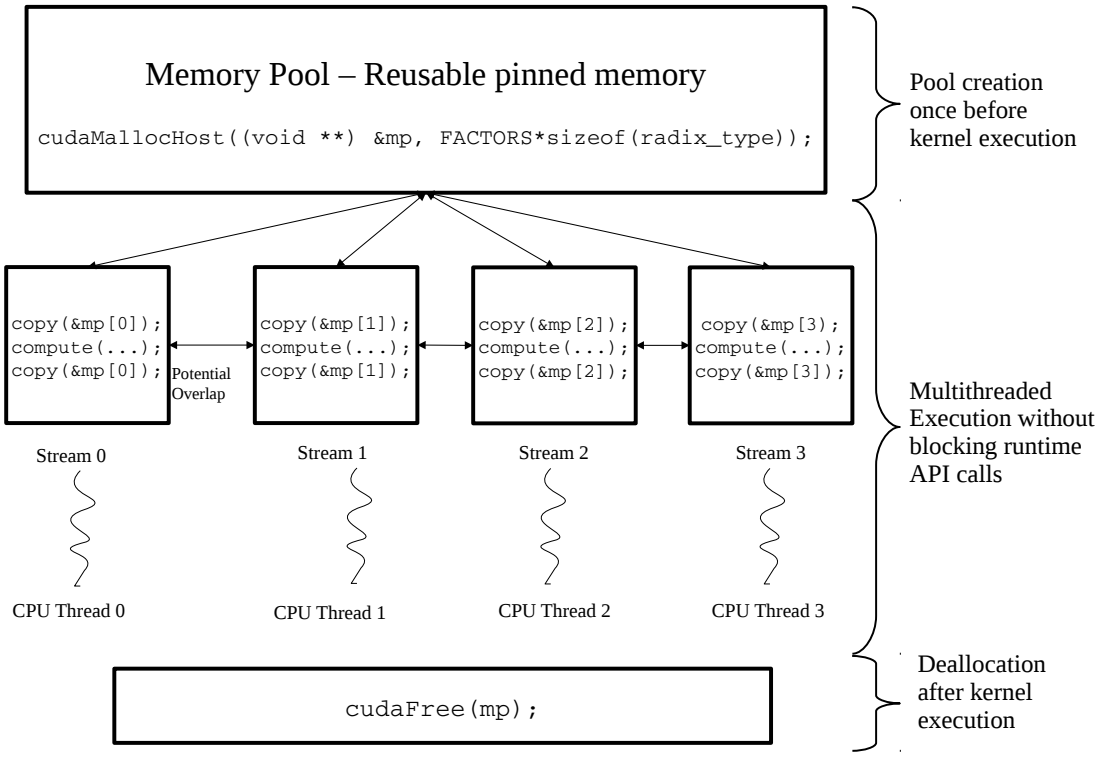


Figure 4.1: Centralized Memory Pool

nature and can also serialize the execution of concurrent CUDA streams. To deal with this, we create a reusable, centralized memory pool which is shared among different host CPU threads. This helps in minimizing the number of smaller CUDA runtime calls into one single call which avoids additional non-trivial runtime overheads associated with such API calls. The size of the memory pool is governed by various factors namely the maximum size of a request batch, maximum number of streams, RSA bitlength, etc. The memory is allocated once at the start of the process. Pinned [26] memory is used to prevent the runtime overheads of creating a staging non-pageable memory area before data transfer can take place between the host and device. The concept of Zero-copy [27] is also used so that device can directly access the host pinned buffer as and when required during kernel execution. However, this may incur an additional overhead in systems where the memory bandwidth of the *PCI* bus is low or limited by features such as *ECC* [28]. In such a situation, it is better to transfer data to the global memory of the device using asynchronous memory copy on a CUDA before processing and from global memory to host after execution in an overlapping fashion to hide the transfer latency. The figure 4.1 illustrates this idea pictorially.

4.3 Optimization 3: Shared Memory

Various types of memories exist in *CUDA memory hierarchy* [29] depending upon the location, program scope and other characteristics. Choosing one type of memory over another other can have a huge impact on the latency and hence it is very important to select the most appropriate memory while designing parallel algorithms. In case the data needs to be shared among threads in a block, *shared memory* is the most convenient

and fastest memory available. In case, the data needs to be accessed in a read-only manner and remains constant during the execution, *texture memories* offer a fast way to store such data which could later be accessed through per-block texture caches. The characteristics of various types of memories is given below which can help while making decision about data residency.

- *Global memory* – slow, off-chip, hundreds of memory cycles.
- *Local memory* – slow, off-chip, difficult to share results among threads.
- *Registers* – fastest, but not suitable for contiguous arrays.
- *Shared memory* – fast, on-chip, suitable for result sharing among threads, multiple banks- parallel R/W, programmer managed cache.
- *L1 cache* – part of shared memory, configurable size, LRU, no control over data residency(works on temporal locality principle).
- *Texture and Constant memory* – read-only memories, perform better with data locality due to caching in SM, Serial Reads if different threads(in warp) access different addresses.

After careful assessment, we use shared and global memory for our use case. However, since the global memory is off-chip, it can amount for a substantial part in the overall latency. Therefore, to minimize the number of global memory loads and stores, we copy the data from global memory to on-chip shared memory once at the start of the kernel execution. Similarly, the result is written back to the global memory from shared memory once at the end of kernel execution. This results in the optimum number of global loads and stores. This results into a common structure for the CUDA kernel design as shown below.

```
//Optimal number of global load/stores
__global__ kernel(...){
        __shared__ shared_memory[];
        //read data from GM once at start
        coalesced_read_from_global_to_shared();
        ....
        compute();
        ....
        //write data to GM once at end
        coalesced_write_back_to_global();
}
```

4.4 Optimization 4: Data Alignment, Access Patterns and Power

We know that the execution of instructions on a GPU takes place in a *SIMD* fashion where a group of threads called a *warp* executes the same instruction inside a block in a SM. Similar to this execution strategy, multiple global memory accesses like loads and stores can also be grouped together as into a single *memory transaction*. This is also known as *Memory Coalescing*. Currently, CUDA supports 32, 64&128 [25][32] byte

transactions. To reduce the number of such memory transactions, we need to make sure that threads in a warp access memory locations that are not far apart. Ideally, the threads should access consecutive global memory locations. However, due to striding, this may not be always possible. To understand the importance of memory coalescing, let us consider the lines of code below which stores some results from shared memory into global memory.

```
res[2*k*bx+2*x] = inter_buf[2*x];
res[2*k*bx+2*x+1] = inter_buf[2*x+1];
```

In the above example, all threads in a warp write to memory locations separated by a distance of two places i.e. a stride of 2. Then the total number of memory transactions will be $ceil(2*32*sizeof(int)/128)$. A modified version of the code with the same effect is shown below.

```
res[2*k*bx+x] = inter_buf[x];
res[2*k*bx+k+x] = inter_buf[x+k];
```

In the above modified version, the threads in a warp write to consecutive memory locations. The total number of memory transactions in this case will be $ceil(32 * sizeof(int)/128)$. Therefore, the number of memory transactions is reduced by half in this case. In addition to global memory coalescing, it is also essential to ensure that there are no shared memory *bank conflicts*. Shared memory is divided into equi-sized memory modules called *banks*. The data is distributed such that consecutive words are located in consecutive banks. If each thread in a warp, access consecutive words, there are no bank conflicts and the shared memory bandwidth is maximum. Therefore, we designed the algorithm such that the shared memory accesses are contiguous. The power drawn by GPUs also depend upon the number of off-chip memory accesses [13]. Therefore, this optimization also results in lower power consumption and hence the cost per decryption is reduced as well.

4.5 Optimization 5: Grid Dimensions and Launch Parameters

The *occupancy* of GPU is an important factor impacting the throughput. It is important to ensure that the GPU is kept busy most of the time. While invoking the kernels, the grid dimensions should be chosen such that there are sufficient threads per block so that each SM has enough number of warps that can be scheduled to hide global memory access latencies and latencies related to other stalls like instruction dependencies, etc. To determine the optimal number of threads per block for each CUDA kernel used, we use a test subroutine as shown below. This subroutine uses sample test data and calculates the most appropriate(lowest latency) grid dimension for a particular kernel.

```
//Determine optimal grid dimensions for each
//kernel by running a few tests
int THREADSPERBLOCK[NUMKERNELS];
function determine_grid_dims(){
        For i=1 to NUMKERNELS do
                THREADSPERBLOCK[i] = optimal_tpb();
}
function optimal_tpb(){
```

```
        Int test_data[];
        Int dim = 2, min_tpb = 2, latency = MAX;
        While dim<=1024 do
                latency = kernel<<<dim>>>(test_data);
                dim*=2;
                If latency<min
                        min_tpb = dim;
        return min_tpb;
}
```

The figures 4.2, 4.3, 4.4, 4.5 show the effect of varying the threads per block or grid dimensions on latency for various CUDA kernels used in the implementation.

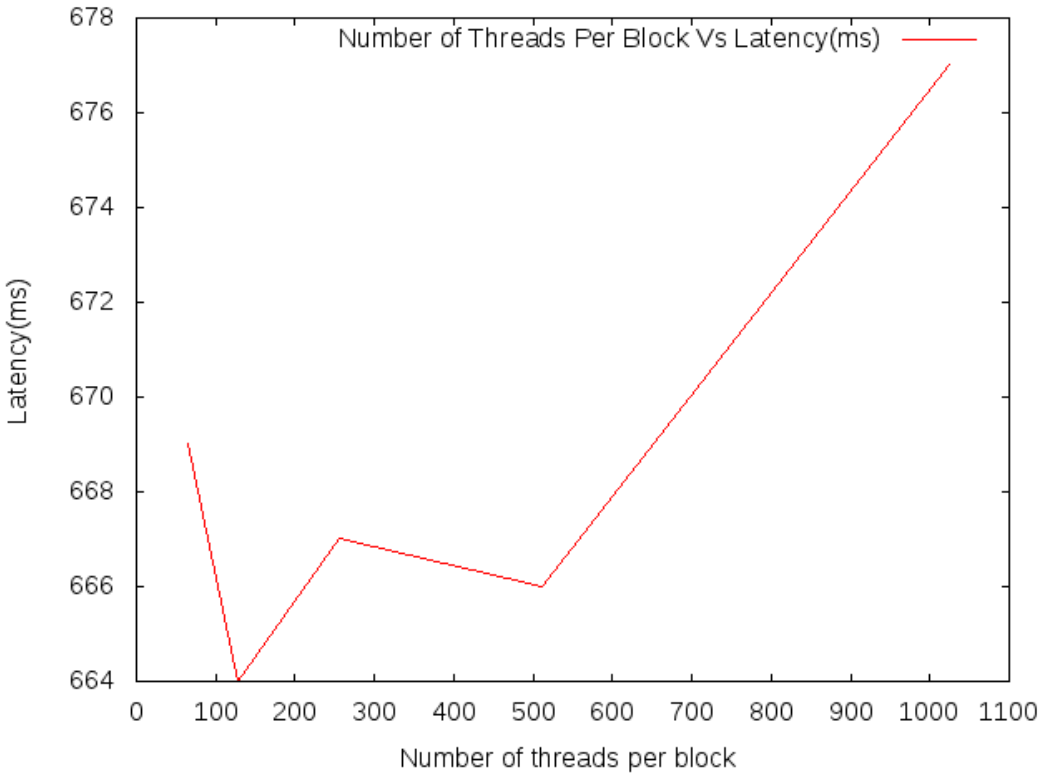


Figure 4.2: Effect of varying grid dimensions on *Add* kernel

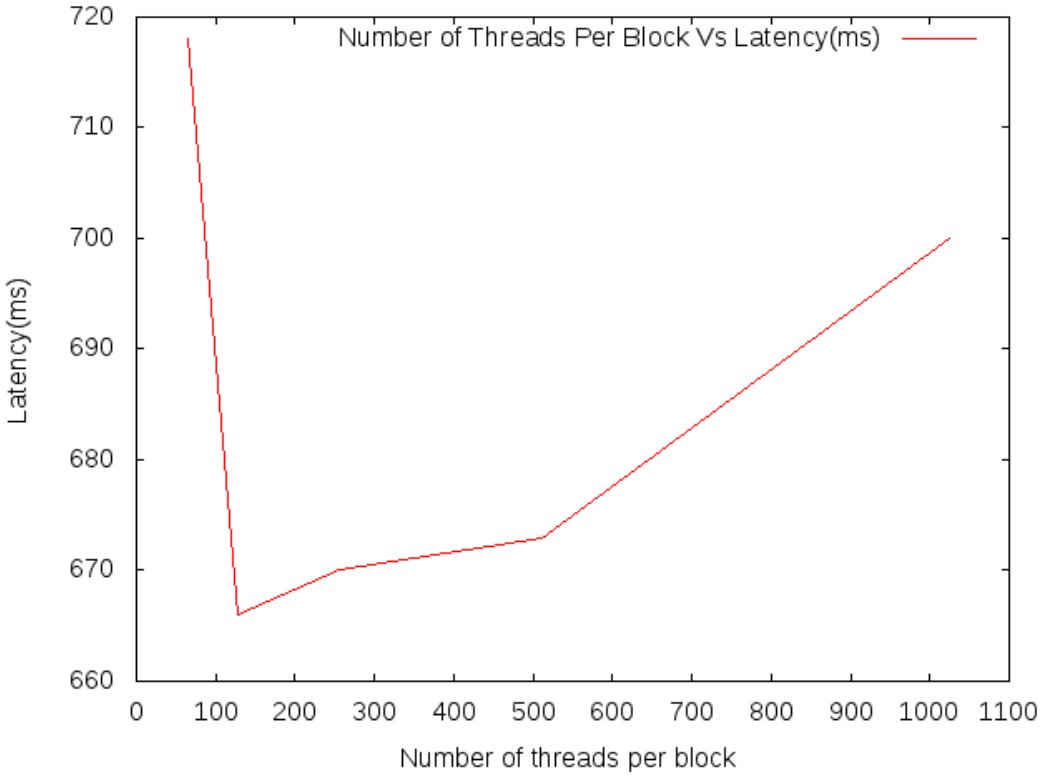


Figure 4.3: Effect of varying grid dimensions on *Mul* kernel

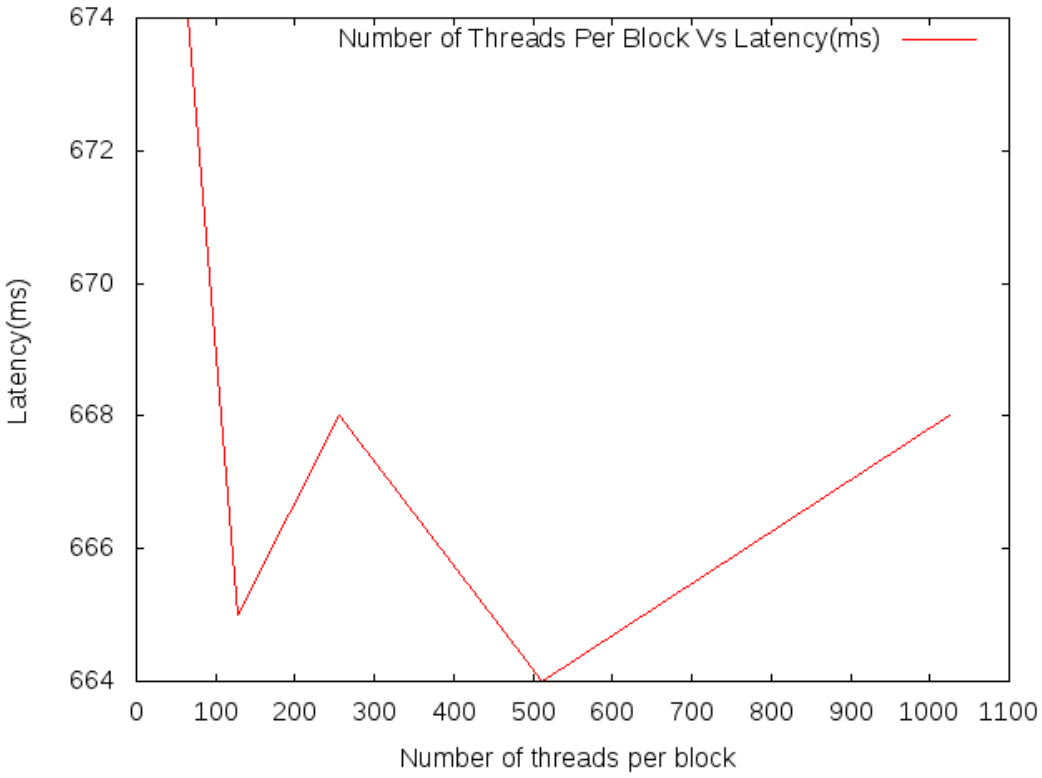


Figure 4.4: Effect of varying grid dimensions on *Sub* kernel

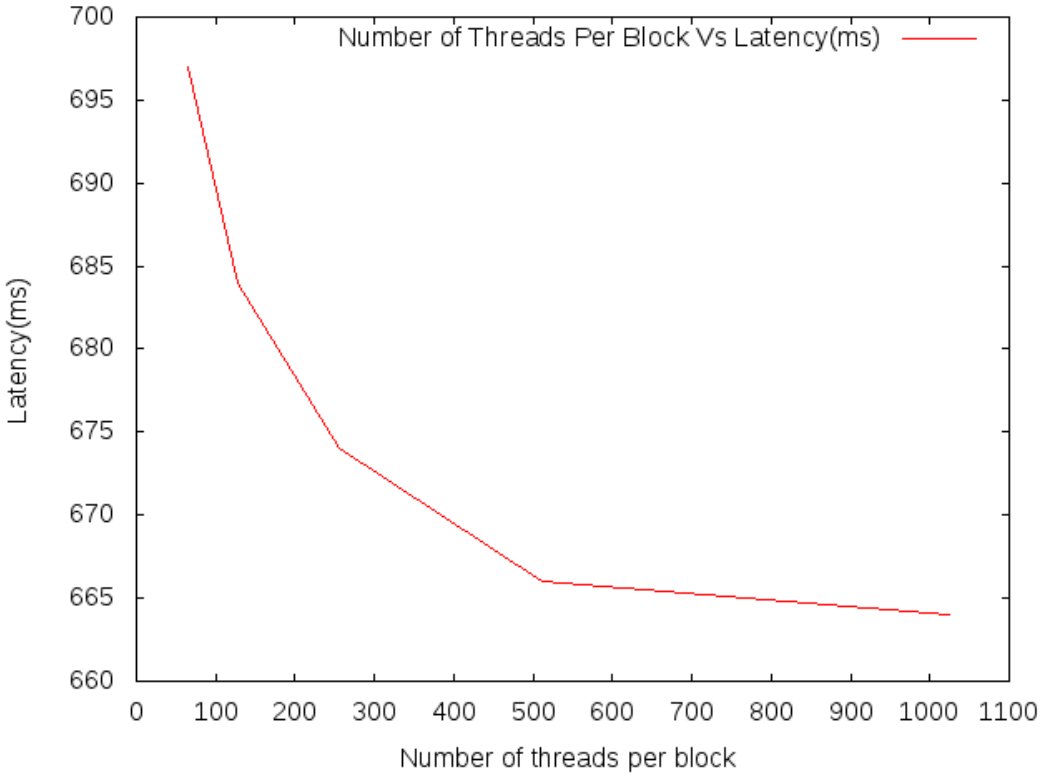


Figure 4.5: Effect of varying grid dimensions on *Rshift* kernel

4.6 Optimization 6: Asynchronous Data Transfer And Execution

As discussed in the adopted heterogeneous model in Implementation 5, the data from device needs to be copied over to the host memory. Synchronous copy will lead to wasted GPU cycles. Since a typical CUDA enabled device consist of multiple copy and execution engines, we use separate CUDA streams for copying the data from device to host and execution of kernels on the device. This will lead to an overlap of kernel execution and data transfer thereby reducing the latency. In addition to this, since we are using multiple host threads, each using a different stream, the execution in the streams is also occurring in an overlapped manner.

4.7 Optimization 7: Higher Base

Initially, we used 2^{16} as the base to represent the input RSA parameters so that the coefficients fit in a 32-bit integer and all the intermediate calculations do not exceed 32 bits. In this case, the length of the parameters is 32 since to represent a 512 bit number in base 2^{16} the number of coefficients will be 32. However, since the complexity of the program depends largely on this length, reducing the length by using a higher base greatly reduces the latency and a higher throughput can be achieved. We changed the base to 2^{32} and used *long* integer type of width 64 bits. Hence, the length of the input parameters is reduced by half to 16 coefficients. As a result, there was around 22.3% improvement in the latency of a single decryption and the throughput increased by more that double from 4,000 to 9,600 decryption per second.

4.8 Optimization 8: Other Optimizations

Apart from the CUDA specific optimizations, the following general optimizations were also incorporated.

- Loop-invariant Code Motion: Any instruction whose behaviour does not depend on the side effects of previous loop iterations can be moved outside the loop.
- Loop Unrolling: The compiler can be directed to unroll for loops so that the overheads of instructions that control the loop. This also presents opportunities for parallel execution of independent instructions present in successive loop iterations. We observed that unrolling a loop where the number of iterations is known at compile time performed better than unrolling the same loop when the number of iterations was not known at compile time and could only be deduced at runtime.
- Common Sub-expression Elimination: A sub-expression which is common among multiple expressions or statements can be evaluated once and the value can be substituted in the associated expressions or statements.
- Function Inlining: The functions with moderate code size can be declared as *inline* to avoid the overheads of a function call.
- NUMA-aware CPU binding: We know that in a Non-Uniform Memory Access(NUMA) architecture, the memory access time depends upon the distance of the memory

location from the processor. Generally, a NUMA architecture consist of multiple NUMA nodes where each node consist of a subset of available processors. A processor, in one NUMA node, accessing non-local memory or shared memory of another NUMA node may incur an additional memory access latency. Therefore, for our multi-threaded application, we would like to use the processors belonging to the same NUMA node. Utilities like *numactl* [42] exist through which one can know the NUMA configuration as well as use specific processors to run the application by physically binding the CPUs to the process.

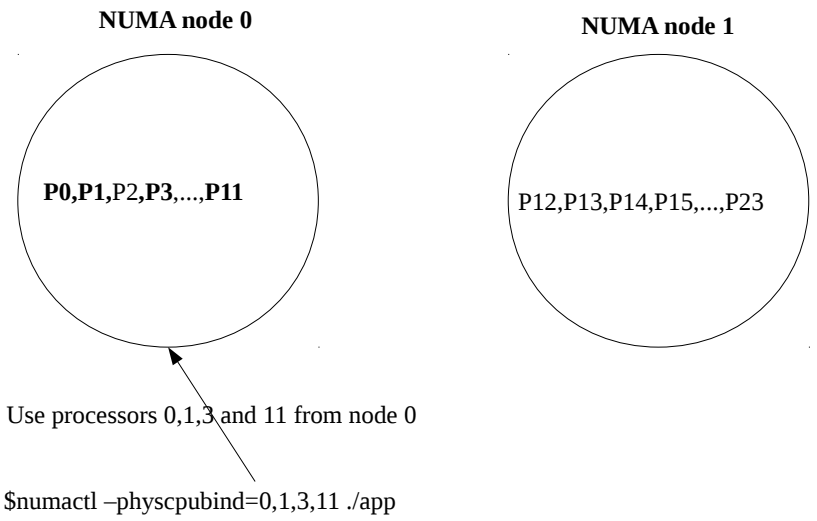


Figure 4.6: NUMA-aware CPU binding

Chapter 5

Observations & Results

In this chapter, we discuss about various experiments that were performed along with the analysis of the effects of various parameters and factors on latency and throughput.

5.1 Experimental Setup

The experiments were performed on the IITD HPC cluster. Each compute node has a couple of *Nvidia Tesla K40* GPUs and a processor with the following specifications.

```
Architecture:          x86_64
CPU op-mode(s):        32-bit, 64-bit
Byte Order:            Little Endian
CPU(s):                24
On-line CPU(s) list:   0-23
Thread(s) per core:    1
Core(s) per socket:    12
Socket(s):             2
NUMA node(s):          2
Vendor ID:             GenuineIntel
CPU family:            6
Model:                 63
Stepping:              2
CPU MHz:               2497.351
BogoMIPS:              4993.68
Virtualization:        VT-x
L1d cache:             32K
L1i cache:             32K
L2 cache:              256K
L3 cache:              30720K
NUMA node0 CPU(s):     0-5,12-17
NUMA node1 CPU(s):     6-11,18-23
```

In addition to this, experiments were also performed on two other GPUs - *GTX690*and *GT730*.

5.2 Latency & Throughput Measurements

As discussed in the Solution section, a number of approaches and implementations were used for the decryption of the pre-master secret. This includes the following different solutions.

- **Version 1:** Multithreaded CPU version where each thread spawns a python process. Data is passed to the process through *pipes* [40].
- **Version 2:** Multithreaded, multi-streamed GPU version using parallel operations like *parMul, parAdd, parSub & parShift*, parallel Montgomery multiplication *parMontgomery* and *square-and-multiply* algorithms.
- **Version 3:** Multithreaded, multi-streamed heterogeneous GPU-CPU version where a portion of operations like addition, subtraction and multiplication are executed on the CPU within the window between kernel invocation and subsequent stream synchronization as shown in the fig 5.1. The execution on CPU is also made multithreaded using OpenMP.

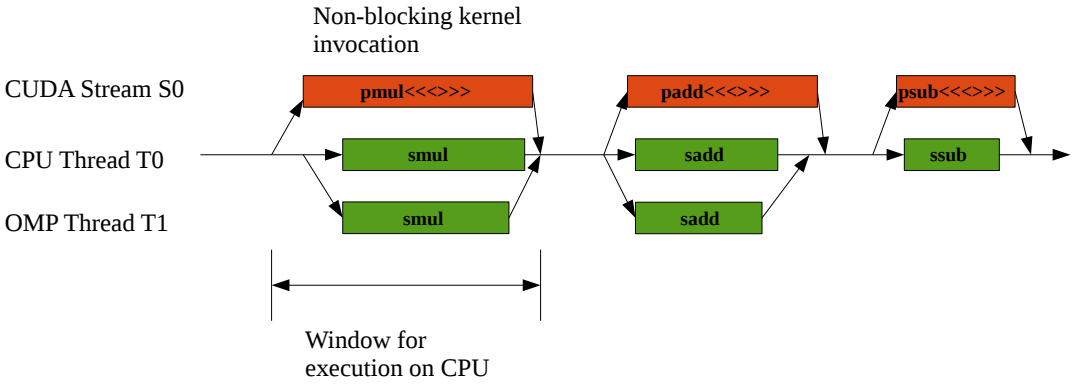


Figure 5.1: Version 3

- **Version 4:** Multithreaded, multi-streamed heterogeneous GPU-CPU version consisting of versions 2 & 3 as well as separate host threads invoking python processes to increase the throughput.
- **Version 5:** Multithreaded CPU version using a serial implementation of the algorithms.

In addition to the above versions, horizontally scalability can be achieved with a MPI solution as discussed in Solution chapter though the latency may increase depending upon the network bandwidth for communication between the nodes.

Table 5.1: Latency

Number of Decryptions(1024-bit)	Latency(ms)		
	Version 1	Version 2	Version 3
1	40	222	223
8	40	227	226
16	59	227	232
32	94	236	249
64	168	253	270
128	311	276	324
256	605	253	318
512	1180	291	409
1024	2344	424	657

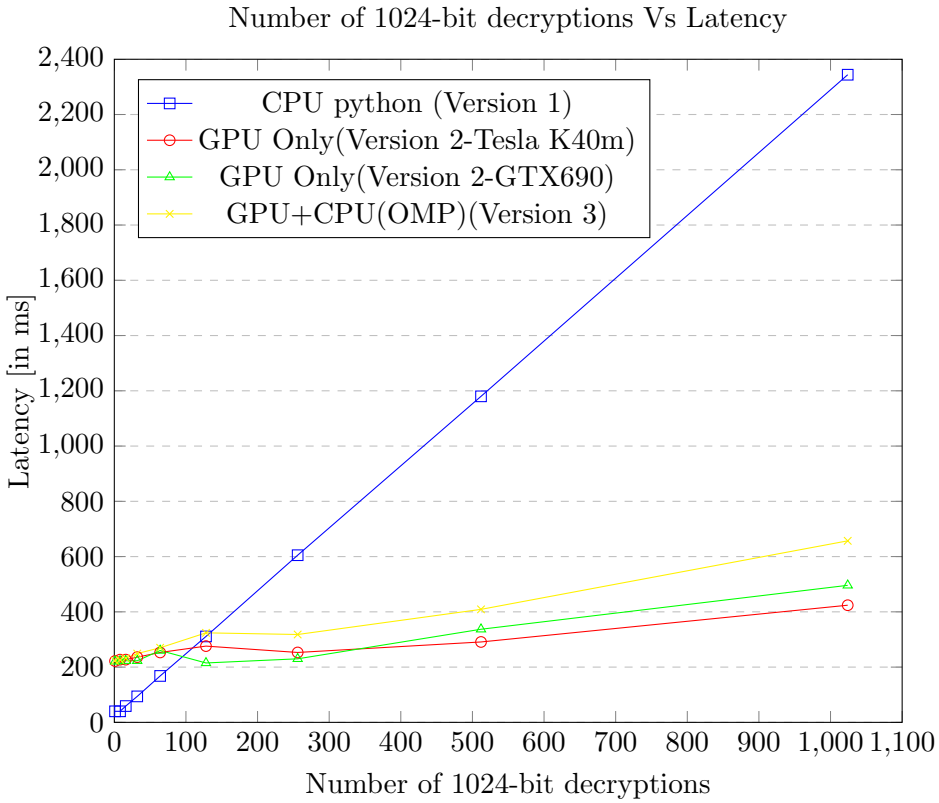


Figure 5.2: A Plot of Number of Decryptions Vs Latency

The Fig shows the effect of number of 1024-bit decryption requests or batch size on latency for various versions. As expected, version 1(execution only on CPU) performed

better for lower number of requests than versions 2 and 3 (GPU) since there are no runtime overheads of kernel launches and data transfers from host to device and device to host. However, as the batch size increases, the latency also increases steadily for version 1 since there are limited number of CPU sockets and cores. We used 8 host CPU threads for the execution of experiments using version 1. On the other hand, in versions 2 and 3, the latency did not rise that much with the increase in the batch size due to the fact that GPUs are many-core throughput architectures and can accommodate more requests with lesser penalty on the latency as long as the number of requests are less than the number of available resources on the GPU. For version 2, a single CUDA stream and a single host CPU thread was used on one K40m graphics card. The maximum throughput observed was *3072 decryption per second per stream*. For version 3, again a single host CPU threads corresponding to a single CUDA stream was used but a portion(50%) of addition operations was executed by 4 OpenMP threads while the kernel was invoked on the GPU.

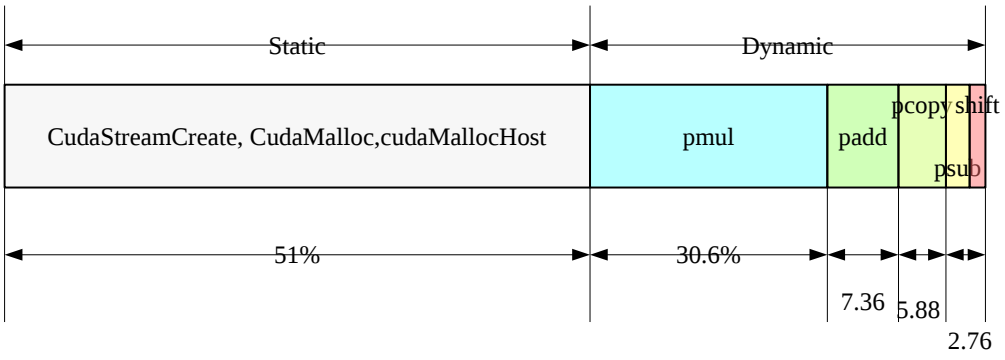


Figure 5.3: Latency Profile of a single 1024-bit decryption

The Fig 5.3 shows the breakdown of latency for a single 1024-bit decryption. The static part corresponds to the activities like CUDA streams creation, memory pool creation on host(pinned) and the device,etc. which are performed during the application context setup. This part does not depend upon the number of requests or batch size. The dynamic part corresponds to the kernel calls which vary with the number of decryptions as discussed above.

Table 5.2: Throughput

Version	Throughput (1024-bit decryptions per second(*dps*))
Version 1	800
Version 2	8192
Version 3	9600
Version 4	10000

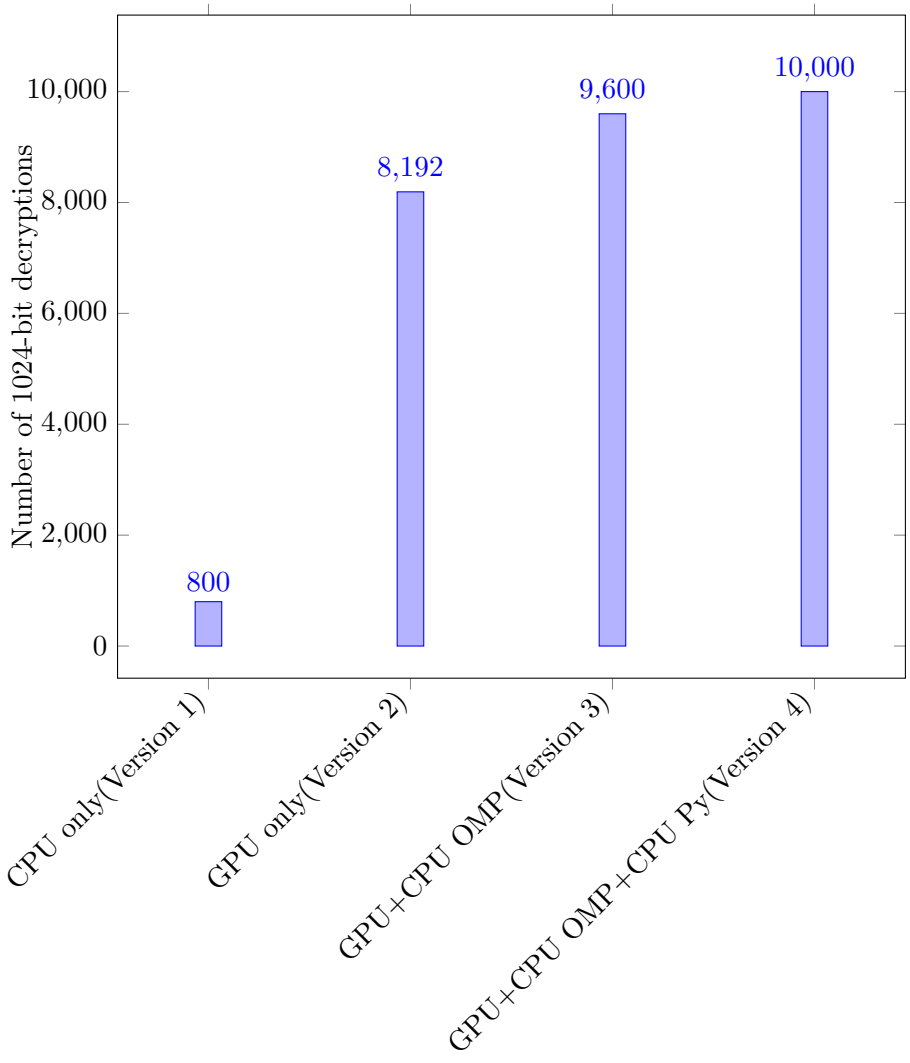


Figure 5.4: Throughput Measurement

The Fig 5.4 shows the peak throughput observed for various versions. Maximum throughput of 10,000 decryptions per second was observed for version 4. 8 CUDA streams were set up each processing 1200 decryption requests and 8 CPU threads each processing 50 requests. With only GPU serving the decryption requests(version 2) the maximum throughput observed was 9600 *dps*. In versions 3 and 4 additional host CPU threads were used to process some requests and operations on CPU side. However,

the throughput is limited by the number of available CPU cores on the host machine. Throughput can further be increased using multiple nodes using an MPI interface. We successfully tested decryption of 16k requests on the IITD HPC cluster [20] containing 4 nodes each node having 8 processors and 2 Tesla K40 GPUs using OpenMPI.

5.3 Effects on Other Parameters

We enumerate below the effects of the development iterations discussed in Solutions chapter on performance critical parameters which were collected using the Nvidia Profiler [41].

1. **Memory Bandwidth**: The parallel algorithms for RSA decryption discussed in this thesis are inherently compute intensive. The amount of data to be transferred from the host to device and from device back to the host is not much. Furthermore, this data transfer latency can be hidden using CUDA streams as discussed in the Optimizations chapter. However, the placement of data within the device plays a crucial role in the overall performance of the application. The threads in a warp may repeatedly access memory for computation. Therefore, data accessed from off-chip global memory and on-chip shared memory or caches can lead to a vast difference in latency. We have used minimal number of global memory accesses by using shared memory accesses. The shared memory load throughput and store throughput were observed to be $874.77GB/s$ and $720.61GB/s$ respectively. On the other hand, the global memory load throughput and store throughput were much lower, $7.17GB/s$ and $7.17GB/s$, highlighting the clear advantage of using shared memory.

2. **Memory Load-Store Efficiency**: Along with data placement, data access patterns is also critical. As discussed before, memory coalescing results into minimum number of global memory transactions and a better bus bandwidth utilization. It is also essential to ensure while accessing shared memory, the threads in a warp should access words stored in different shared memory banks to prevent shared memory bank conflicts. We have incorporated global memory coalescing and shared memory bank conflict prevention strategies as described in the Optimizations sections. The profiler reported global memory load and store efficiency as 100% due to memory coalescing.

3. **Execution Efficiency**: A number of CUDA profiler metrics are present which can indicate the execution efficiency of the kernels. The profiler reported the *IPC* (instructions executed per clock cycle) as 1.66 which is essentially the ratio of the number of instructions executed per SM per clock cycle. The eligible warps per active cycle was reported as 3.94 which indicates the number of warps that can execute their next instruction and hence can be scheduled next by the warp schedulers. The multiprocessor activity(i.e. the ratio of time atleast one warp is active on any SM to the total time) was around 88%. The warp execution efficiency was around 72%. Please note that the most of control-flow code consist of only a single *if* statement without an *else* clause. Hence, there is no serialization of multiple if-else branches.

4. **Power**: One of the factors on which the cost per decryption depends upon is the power consumption of the GPUs. The power consumption in turn depends

upon the number of off-chip global memory accesses as discussed in [13]. The number of global memory accesses were reduced by copying the data from global memory once to the on-chip shared memory at the beginning of the computation and writing the results back once at the end. Using this approach the power consumption reduced by around 13% from $88.7W$ to $77.2W$.

5.4 Comparison with SSLShader[1]

Keon Jang et. al [1] proposed a highly optimized solution achieving a peak throughput of 27,000 RSA transactions per second. Our implementation is an implementation as well as an augmentation of their proposed work. In addition to devising and developing the technical implementation from scratch, we proposed additional optimization techniques and used a heterogeneous environment utilizing the available CPU cores on the same node as well as connected nodes via an MPI interface. The novel optimizations techniques we proposed are as follows. We proposed memory pooling technique so that overheads of memory allocation on device and host can be reduced and same memory can be reused. This also ensures that other CUDA streams do not get blocked as memory allocation on device blocks all the independent streams. This latency of creation of memory pool on host and device contributes to the static portion of the overall latency as discussed in the previous section. Another optimization we discussed is reducing the overall number of global load/stores by using shared memory and accessing the global memory only once at the start of the kernel for reading data and then writing the results back to the global memory once at the end in a coalesced fashion. In our experiments, this led to an improvement of 43% in latency of a single 1024-bit decryption. We also analyzed the effect of global load/stores on power consumption and observed a reduction of 13%in the power consumption as the number of off-chip access are lesser. We proposed empirical analysis to determine the optimal grid dimensions as well as the task sharing ratio between CPU and GPU by performing some tests at the start of the application. This led to an increase of 17% in the throughput. In addition to these optimizations, our solution is horizontally and vertically scalable and can leverage the availability of commodity hardware to a good extent by utilizing the available CPUs and GPUs on the same machine or a cluster of connected machines.

5.5 Throughput Comparison - K40m Vs GTX690 Vs GT730

We have performed experiments on 3 different GPUs - K40m, GTX690 and GT730. While K40m and GTX690 are quite powerful with 2,880 and 3,072 CUDA cores and typically used in high-end servers and high performance computing environments, GT730 is a mid-range regular graphics card with 384 cores. Therefore, the throughput obtained on GT730 is much lower as compared to K40m and GTX690. The peak throughput obtained on a node containing 2 K40m GPUs was 10,000 *dps* where 16 CPU threads were used. This node had a xeon CPU with 2 sockets with each socket having 12 cores. GTX690 which contains two GTX680 cards on the same board offered a peak throughput of 5,056 *dps*, The throughput was limited by the number of cores on the machine which was only 4. The throughput on GT730 using a single thread and a single CUDA stream was 512 *dps*.

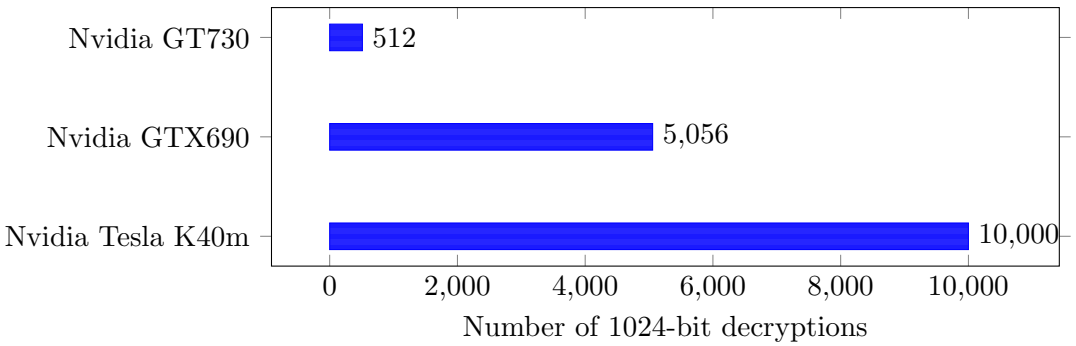


Figure 5.5: Throughput Comparison of Different GPUs

Chapter 6

Conclusion

In this thesis, we proposed a heterogeneous system to accelerate SSL decryption at the server side and increase the capacity of the server to handle more SSL clients. We explored various optimizations related to exploitation of data parallelism in algorithms used for fast modular exponentiation, efficient memory management techniques like memory pooling, efficient memory access patterns for coalescing and avoiding shared memory bank conflicts, maximizing device occupancy by calculating optimal grid dimensions, asynchronous memory copy and kernel execution using streams and other useful optimizations. We also looked into the effect of global memory load-stores on the power consumption and reduced the power drawn by 13% using minimal number of global memory accesses and memory coalescing. We tried to maximize the overall system throughput by utilizing the available CPU cores using OpenMP and pthread libraries. We also incorporated the use of message passing interface like OpenMPI to divide the work among available nodes in a cluster. Hence, our solution is both vertically and horizontally scalable. Solutions have been proposed by reseachers in [16] [17] [18] achieving a throughput of 1800, 3863 and 5856 decryptions per second respectively. We achieved a maximum throughput of **10000** 1024-bit RSA decryptions per second on a single node containing two K40 GPUs and a xeon CPU. Faster optimized commercial solutions like *SSLShader* [1] exist which offer a peak throughput of 27k transactions per second.

Chapter 7

Future Work

This work present an opportunity to explore more optimizations to improve the throughput even further. We have used an empirical approach to divide work among multiple CPUs and GPUs such that the overall throughput is maximized. A dynamic load balancing scheme has been explored by *L. Chen et al.* using the concept of *persistent kernels* [14] which can serve as an useful augmentation of this work. We touched upon using OpenMPI to run the application on a cluster of compute nodes. The strategies to divide work among nodes by the master node and the effects of factors like network bandwidth on latency can be studied.

Bibliography

[1] Keon Jang et. al, *Sslshader: Cheap SSL Acceleration With Commodity Processors*, Proc. of the 8th USENIX Conference on Networked systems and Implementation, NSDI11.

[2] J.-J. Quisquater and C. Couvreu, *Fast Decipherment Algorithm for RSA Public-key Cryptosystem*, Electronics Letters, 18(21):905907, 1982.

[3] P. Montgomery, *Modular Multiplication Without Trial Division*, Mathematics of Computation, 44(170):519521, 1985.

[4] Maged Michael et al. *Scale-up x Scale-out: A Case Study using Nutch/Lucene.* Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International.

[5] V. Bunimov, M. Schimmler, and B. Tolg, *A Complexity-Effective Version of Montgomerys Algorithm*, Workshop on Complexity Effective Designs, ISCA02, May 2002.

[6] Szerwinski, R., Gneysu, T., *Exploiting the Power of GPUs for Asymmetric Cryptography*, In: CHES 2008 [39], pp. 7999 (2008).

[7] Yang Y, Li C, Zhou H , *CUDA-NP: Realizing Nested Thread-level Parallelism in GPGPU Applications*, J Computer Science Technology 30(1):319.

[8] M. Shand and J. Vuillemin, *Fast Implementation of RSA Cryptography*, 1th IEEE Symposium on Computer Arithmetic, Windsor, Ontario, Canada, pp. 252-259, 1993.

[9] J. Gilger, J. Barnickel, and U. Meyer *GPU-acceleration of Block Ciphers in the OpenSSL Cryptographic Library*, Proc. of the 15th int. conf. on Information Security, ser. ISC12. Springer-Verlag, 2012, pp. 338353.

[10] R Landaverde et. all, *An investigation of Unified Memory access performance in CUDA*, Extreme Computing Conference (HPEC), 2014 IEEE, 2014.

[11] Seluk, B. and Erkay, S, *Highly-Parallel Montgomery Multiplication for Multi-core General-Purpose Microprocessors*, Computer and Information Sciences III, 27th International Symposium on Computer and Information Sciences, pp. 467-476, 2013.

[12] Neves, S. and Araujo, F., *On the Performance of GPU Public-Key Cryptography*, IEEE International Conference on Application-Specific Systems, Architectures and Processors (ASAP), pp. 133-140, 2011.

[13] S. Collange, D. Defour, and A. Tisserand, *Power consumption of gpus from a software perspective*, Computational Science, pages 914923, 2009.

[14] L. Chen, O. Villa, S. Krishnamoorthy, and G. Gao, *Dynamic Load Balancing on Single- and Multi-GPU Systems*, Proc. of the IEEE International Parallel and Distributed Processing Symposium, 2010.

[15] P. Xiang, Y. Yang, M. Mantor, N. Rubin and H. Zhou, *Revisiting ILP Designs for Throughput-Oriented GPGPU Architecture*, 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, May, 2015.

[16] H. M. Fadhil and M. I. Younis, *Parallelizing rsa algorithm on multicore cpu and gpu*, International Journal of Computer Applications, 87(6):1522, February 2014. Published by Foundation of Computer Science, New York, USA. 11.

[17] Z. Yao, V. Chenghua, C. Yuwei, and L. Wei, *Efficient Acceleration of RSA Algorithm on GPU*, IEEE International Conference on Oxide Materials for Electronic Engineering (OMEE), Lviv, Ukraine, pp. 531-534, 03 - 07 September, 2012.

[18] M. I. Younis, H. M. Fadhil, Z. N. Jawad, *Acceleration of the RSA Processes based on Parallel Decomposition and Chinese Reminder Theorem*, International Journal of Application or Innovation in Engineering and Management, Volume 5, Issue 1 January, 2016.

[19] https://en.wikipedia.org/wiki/SSL_acceleration

[20] http://supercomputing.iitd.ac.in/

[21] http://developer.download.nvidia.com/compute/cuda/CUDA_Occupancy_calculator.xls

[22] http://devblogs.nvidia.com/parallelforall/unified-memory-in-cuda-6/

[23] http://devblogs.nvidia.com/parallelforall/how-overlap-data-transfers-cuda-cc/

[24] https://devblogs.nvidia.com/parallelforall/using-shared-memory-cuda-cc/

[25] https://devblogs.nvidia.com/parallelforall/how-access-global-memory-efficiently-c

[26] https://devblogs.nvidia.com/parallelforall/how-optimize-data-transfers-cuda-cc/

[27] http://docs.nvidia.com/cuda/cuda-c-best-practices-guide/#zero-copy

[28] https://en.wikipedia.org/wiki/ECC_memory

[29] http://docs.nvidia.com/cuda/cuda-c-programming-guide/#memory-hierarchy

[30] http://docs.nvidia.com/cuda/

[31] http://developer.download.nvidia.com/assets/cuda/files/CUDADownloads/TechBrief_Dynamic_Parallelism_in_CUDA.pdf

[32] http://calcul.math.cnrs.fr/IMG/pdf/CUDA-Optimization-Julien-Demouth.pdf

[33] https://bluewaters.ncsa.illinois.edu/c/document_library/get_file?uuid=38ef0ba5-4d6c-4ecc-82bf-6c2a85ce7188&groupId=10157

[34] http://www.ibm.com/support/knowledgecenter/SSFKSJ_7.1.0/com.ibm.mq.doc/sy10660_.htm?lang=en

[35] https://docs.python.org/2/library/functions.html#pow

[36] http://formalverification.cs.utah.edu/GKLEE/

[37] https://www.open-mpi.org/

[38] https://www.openmp.org/

[39] http://docs.nvidia.com/cuda/

[40] http://www.tldp.org/LDP/lpg/node12.html

[41] http://docs.nvidia.com/cuda/profiler-users-guide/#axzz47PBespr6

[42] http://linux.die.net/man/8/numactl