WROCŁAW UNIVERSITY OF SCIENCE AND TECHNOLOGY
FACULTY OF FUNDAMENTAL PROBLEMS OF TECHNOLOGY

FIELD: Computer Science
SPECIALIZATION: Computer Security

MASTER OF SCIENCE THESIS

Timing Attack Resistant Implementation of RSA on GPU.

AUTHOR:
Krzysztof Hamerski

SUPERVISOR:
dr Maciej Gębala

GRADE:

WROCŁAW 2017

Contents

Chapter 1

Introduction

Public key cryptography is the key factor in providing secure communication between two parties. Fast development of distributed system requiring not only security, but also integrity and non-repudiation has pushed cryptography to the limit. Since 1978[15] most commonly used cryptosystem is RSA, which provides asymmetric encryption, as well as generation of digital signatures. The security of RSA is mainly based on the bitwise key length. As computational power of modern CPUs arises, the minimal bit length of RSA key gets significantly bigger to provide sufficient security. At least 4096 bits long keys are considered secure nowadays. This leads to very high workload required to perform encryption/decryption. RSA is mainly based on modular arithmetics and simple computations become infeasible for moder CPUs, when dealing with so large integers.

One of the solution to this problem is to parallelize. GPGPU[18] (for General-Purpose computing on the Graphics Processing Unit) enables the use of GPU for parallel computation other than graphics. GPUs are designed to perform computations in parallel. Since every PC is equipped with some kind of GPU, one can easily exploit its capabilities. NVIDIA has made it even more accessible by creating CUDA (Compute Unified Device Architecture)[16]. It is a parallel computing platform and API, which exposes GPUs true potential.

1.1 Environment Specification

This project was developed in Microsoft Visual Studio 2015[14] with NSight plugin and CUDA API.[16] Source code is written mainly in C++ and inline assembly - PTX.[17] Table 1.1 more precisely illustrates GPU specification on which the program and all tests are run.

Table 1.1: Device specification

Device name:	GeForce GTX 960
CUDA Driver Version:	9.0
CUDA Runtime Version:	8.0
CUDA Capability version number:	5.2
Total amount of global memory:	4096 MBytes (4294967296 bytes)
GPU Max Clock rate:	1304 MHz (1.30 GHz)
Total amount of shared memory per block:	49152 bytes
Warp size:	32

Table below presents full platform and system information.

Table 1.2: PC specification

Operating System:	Windows 10 Education 64-bit
Motherboard:	Gigabyte Technology Co., Ltd. P55A-UD4
CPU:	Intel(R) Core(TM) i5 CPU 750 @ 2.67GHz (4 CPUs), 2.7GHz
RAM Memory: :	4096MB

Chapter 2

Theory

This chapter basically presents minimal amount of theory required to fully understand the concepts, problems and solutions which were applied and implemented within this project.

2.1 RSA

The RSA algorithm was created by Ronald Rivest, Adi Shamir, and Leonard Adleman in 1970s. The security of the algorithm is based on the problem of integer factorization.

- Key generation[12]
  - Choose two large and distinct prime numbers p and q.
  - Compute the modulus n
    $$n = pq$$
  - Compute
    $$\lambda(n) = lcm(\lambda(p), \lambda(q)) = lcm(p-1, q-1)$$
    where λ is the Carmichael Totient Function[8]. This value must be kept in private.
  - Choose an integer e such that:
    $$1 < e < \lambda(n)$$
    and
    $$gcd(e, \lambda(n)) = 1$$
    This means e and $\lambda(n)$ are coprime.[21]
  - Choose value d for decryption and solve for d:
    $$de \equiv 1(mod\lambda(n))$$
    e is used for encryption. Usually this is done the other way around. e is chosen first so it has short bit length and small Hamming weight,[9] which provides for much faster encryption. In most cases $e = 3$,5 or 7. High security is provided if larger number is used. Most common is Fermat's four: $e = 2^{16} + 1 = 65537 =$ 0x10001

– Public key is then:

$$(e, n)$$

and the private key:

$$(d, n)$$

- Encryption

  For message m the ciphertext c is in relation:

$$c \equiv m^e \bmod n$$

- Decryption

$$m \equiv c^d \bmod n \equiv (m^e)^d \bmod n \equiv m \bmod n$$

2.2 Side channel attacks

Side-channel attacks are very powerful attacks against cryptographic implementations. They take the advantage of implementation flaws, gathering information about almost everything that can be measured during execution time. If no care is taken, side-channel attacks can be used to compromise any mathematically secure system.

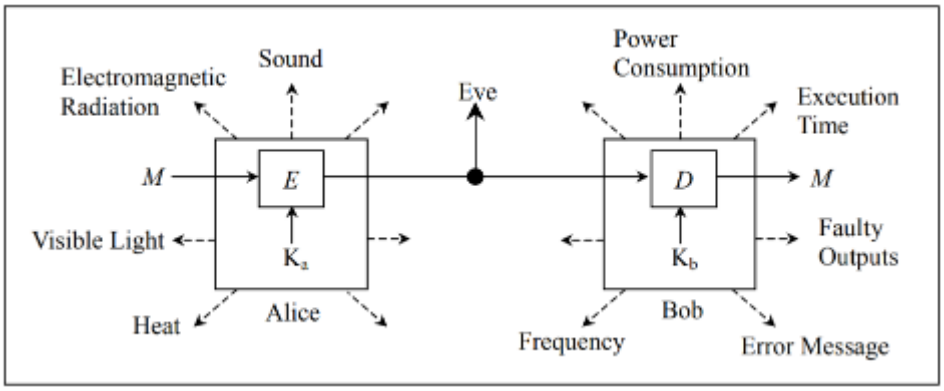

Figure 2.1: An example of cryptosystem including side-channel information leakage

Side-channel attacks aim to retrieve secret data from a cryptographic system by observing factors outside the normal computation. Power consumption, execution time and even electromagnetic radiation, sound, light and heat can leak some information about data being computed. Running statistical analysis on this partial information makes even the most sophisticated systems vulnerable to breakage.

2.2.1 Timing attacks

Implementations of cryptographic algorithms often perform computations in non-constant time, due to performance optimizations. If such operations involve secret parameters, these timing variations can leak some information and, provided enough knowledge of the implementation at hand, a careful statistical analysis could even lead to the total recovery of these secret parameters.[22] Perfect example of the private key leakage is implementation of RSA, using Square and Multiply algorithm.[19]

Algorithm 1 Square and multiply

1: *Input* : $x, e, n \in N$
2: *Output* : $m^e \bmod n$
3: a $\leftarrow 1$
4: **for** $i = k - 1$ to 0 **do**
5: a $\leftarrow a^2 \bmod n$
6: **if** $d_i = 1$ **then**
7: a $\leftarrow a \times m \bmod n$
8: **return** a

The algorithm leaks information about exponent e by analyzing difference in computations when $d_i = 1$, which is presented on the figure below.[3]

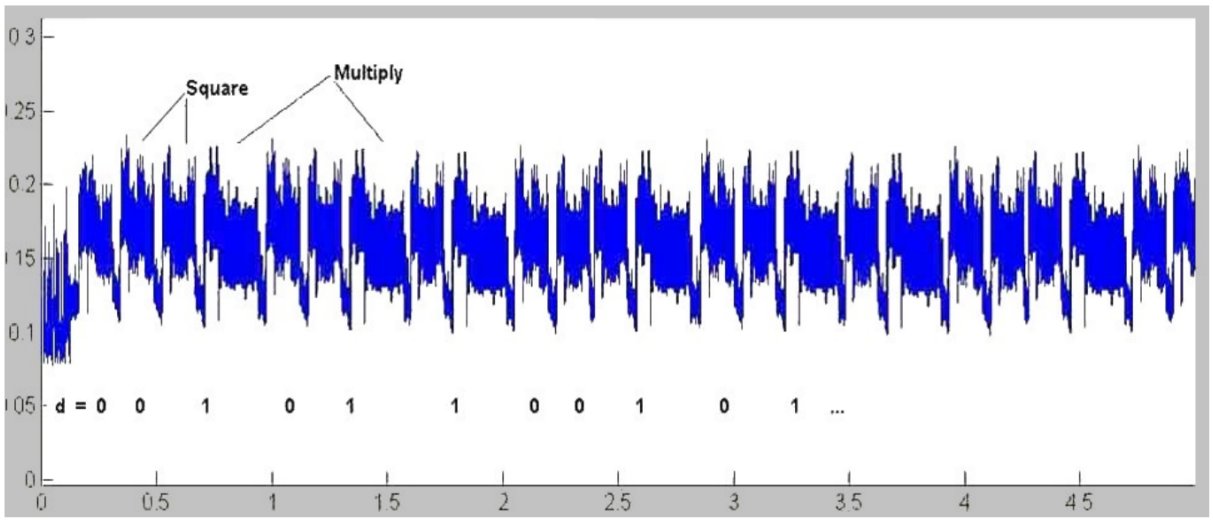

Figure 2.2: Square and Multiply information leakage

2.3 Montgomery Ladder Algorithm

Montgomery Ladder algorithm is a remarkably simple method of computing modular exponentiation, keeping resistance to side-channel attacks. It performs both multiplication and squaring every time the loop is iterated. The complexity of the algorithm is $log_2 n$, where n is the bitwise length of the exponent.[1]

Algorithm 2 Montgomery Ladder Algorithm

1: *Input* : $x, e, n \in N$
2: *Output* : $m^e \bmod n$
3: $R_0 \leftarrow 1$
4: $R_1 \leftarrow$ m
5: **for** $i = k - 1$ to 0 **do**
6: $R_{1-d_i} \leftarrow R_0 \times R_1 \bmod n$
7: $R_{d_i} \leftarrow R_{d_i}^2 \bmod n$
8: **return** R_0

2.4 Parallel programming on CUDA

CUDA (Compute Unified Device Architecture (the acronym dropped recently)) - is a parallel computing platform and API developed by Nvidia. Having CUDA-capable Graphics Processing Unit, the platform enables general purpose computations on the graphics card. CUDA is basically an intermediary between host and the GPU. It provides direct access to device's kernels, instructions and other components for parallel execution.

The calculations on CUDA are executed on SM - streaming multiprocessor. Threads are organized in blocks. A block is executed by a multiprocessing unit. Blocks are further organized into grids. The number of blocks being able to execute simultaneously matches the number of SMs on the device. The threads are also executed simultaneously in a group of 32 called a warp. Threads can also synchronize and share common memory, which is thoroughly explained in the 2.4.1.

2.4.1 Shared Memory

"Threads within a block can cooperate by sharing data through shared memory and by synchronizing their execution to coordinate memory accesses. Because it is on-chip, shared memory has much higher bandwidth and much lower latency than local or global memory. For that shared memory is equivalent to a user-managed cache: The application explicitly allocates and accesses it. A typical programming pattern is to stage data coming from device memory into shared memory, process the data in shared memory while sharing the shared view of the data across the threads of a block, and write the results back to device memory.

To achieve high bandwidth, shared memory is divided into equally-sized memory modules, called banks, which can be accessed simultaneously. Any memory read or write request made of n addresses that fall in n distinct memory banks can therefore be serviced simultaneously, yielding an overall bandwidth that is n times as high as the bandwidth of a single module. However, if two addresses of a memory request fall in the same memory bank, there is a bank conflict and the access has to be serialized. The hardware splits a memory request with bank conflicts into as many separate conflict-free requests as necessary, decreasing throughput by a factor equal to the number of separate memory requests. If the number of separate memory requests is n, the initial memory request is said to cause n-way bank conflicts. To get maximum performance, it is therefore important to minimize bank conflicts. Shared memory has 32 banks that are organized such that successive 32bit words map to successive banks. A shared memory request for a warp does not generate a bank conflict between two threads that access any address within the same 32bit word (even though the two addresses fall in the same bank). For devices of compute capability 3.x, shared memory has 32 banks with two addressing modes that can be configured using cudaDeviceSetSharedMemConfig(). Each bank has a bandwidth of 64 bits per clock cycle. In 64bit mode, successive 64bit words map to successive banks. A shared memory request for a warp does not generate a bank conflict between two threads that access any sub-word within the same 64bit word (even though the addresses of the two sub-words fall in the same bank). In 32bit mode (default), successive 32bit words map to successive banks. A shared memory request for a warp does not generate a bank conflict between two threads that access any sub-word within the same 32bit word or within two 32bit words whose indices i and j are in the same 64word aligned segment (i.e., a segment whose first index is a multiple of 64) and such that j=i+32 (even though the addresses of the two sub-words fall in the same bank)."[7]

Chapter 3

Implementation

The code was written in C++ language. In order to minimize data movement between host and the device, most of logics and computation are executed fully on GPU. Implementation of RSA encryption requires only one function - modular exponentiation of a multi precision integer. Implemented Big Integer class provides much more functions, to properly handle data and validate results. Code listing below shows the header of class BigInteger.

```
class BigInteger
{
//fields
public:

// 4096 bits
static const int ARRAY_SIZE = 128;

private:
// Magnitude array in little endian order.
// Most-significant int is mag[length-1].
// Least-significant int is mag[0].
// Allocated on the device.
unsigned int* deviceMagnitude;

// same array allocated on the host
// provides faster access if nothing was changed
unsigned int* hostMagnitude;

// flag indicating if hostMagnitude matches deviceMagnitude
bool upToDate;

// Device wrapper instance diffrent for every integer
// to provide parallel execution
DeviceWrapper* deviceWrapper;

// methods
public:
BigInteger();
BigInteger(const BigInteger& x);
BigInteger(unsigned int value);
~BigInteger();
const unsigned int& operator[](int index);

// factory
static BigInteger* fromHexString(const char* string);
```

```
static BigInteger* createRandom(int bitLength);

// setters, getters
void set(const BigInteger& x);
unsigned int* getDeviceMagnitude(void) const;

// arithmetics
void add(const BigInteger& x);
void subtract(const BigInteger& x);
void multiply(const BigInteger& x);
void square(void);
void mod(const BigInteger& modulus);
void multiplyMod(const BigInteger& x, const BigInteger& modulus);
void squareMod(const BigInteger& modulus);
void powerMod(BigInteger& exponent, const BigInteger& modulus);

// logics
void shiftLeft(int bits);
void shiftRight(int bits);

// extras
bool equals(const BigInteger& value) const;
int compare(const BigInteger& value) const;
int getBitwiseLengthDiffrence(const BigInteger& value) const;
int getBitwiseLength(void) const;
int getLSB(void) const;
bool testBit(int bit);
void synchronize(void);
char* toHexString(void);
void print(const char* title);

//timer
void startTimer(void);
unsigned long long stopTimer(void);

// async calls
// must call synchronize to read from
void modAsync(const BigInteger& modulus);
void multiplyModAsync(const BigInteger& x, const BigInteger& modulus);
void squareModAsync(const BigInteger& modulus);

private:

void setMagnitude(const unsigned int* magnitude);
void clear(void);
void updateDeviceMagnitiude(void);
void updateHostMagnitiude(void);
static unsigned int random32(void);

/*
Parses hex string to unsigned int type.
Accepts both upper and lower case, no "0x" at the beginning.
E.g.: 314Da43F
*/
static unsigned int parseUnsignedInt(const char* hexString);
};
```

Listing 3.1: BigInteger.h

Big Integer class handles mathematical logic, validates input / output, and provides comfortable and readable interface.

The intermediary between Big Integer and GPU is another class - Device Wrapper. It handles GPU kernel launches, synchronization and data movement. Class definition is listed below.

```
class DeviceWrapper
{

private:

// main stream for kernel launches
cudaStream_t mainStream;

// lauch config
dim3 block_1, block_2, block_4;
dim3 thread_warp, thread_2_warp, thread_4_warp;

// 4 ints to help store results
int* deviceWords;

// auxiliary arrays
unsigned int* device4arrays;
unsigned int* device128arrays;
unsigned int* deviceArray;

unsigned long long* deviceStartTime;
unsigned long long* deviceStopTime;

public:

DeviceWrapper();
~DeviceWrapper();

// sync
unsigned int* init(int size) const;
unsigned int* init(int size, const unsigned int* initial) const;
void updateDevice(unsigned int* device_array, const unsigned int*
    host_array, int size) const;
void updateHost(unsigned int* host_array, const unsigned int*
    device_array, int size) const;
void free(unsigned int* device_x) const;

// extras
void clearParallel(unsigned int* device_x) const;
void cloneParallel(unsigned int* device_x, const unsigned int* device_y)
     const;
int compareParallel(const unsigned int* device_x, const unsigned int*
    device_y) const;
bool equalsParallel(const unsigned int* device_x, const unsigned int*
    device_y) const;
int getLSB(const unsigned int* device_x) const;
int getBitLength(const unsigned int* device_x) const;
void synchronize(void);

// measure time
void startClock(void);
unsigned long long stopClock(void);
```

```
// logics
void shiftLeftParallel(unsigned int* device_x, int bits) const;
void shiftRightParallel(unsigned int* device_x, int bits) const;

// arithmetics
void addParallel(unsigned int* device_x, const unsigned int* device_y)
    const;
void subtractParallel(unsigned int* device_x, const unsigned int*
    device_y) const;
void multiplyParallel(unsigned int* device_x, const unsigned int*
    device_y) const;
void squareParallel(unsigned int* device_x) const;
void squareParallelAsync(unsigned int* device_x) const;
void modParallel(unsigned int* device_x, unsigned int* device_m) const;
void modParallelAsync(unsigned int* device_x, unsigned int* device_m)
    const;
void multiplyModParallel(unsigned int* device_x, const unsigned int*
    device_y, const unsigned int* device_m) const;
void multiplyModParallelAsync(unsigned int* device_x, const unsigned int
    * device_y, const unsigned int* device_m) const;
void squareModParallel(unsigned int* device_x, const unsigned int*
    device_m) const;
void squareModParallelAsync(unsigned int* device_x, const unsigned int*
    device_m) const;

private:
void inline addParallelWithOverflow(unsigned int* device_x, const
    unsigned int* device_y, int blocks) const;
};
```

Listing 3.2: DeviceWrapper.h

Project also contains Test class to validate computations, measure times and simulate encryption. RSA class contains single "encrypt" function, which encrypts provided value. Full class diagram is presented on figure 3.1.

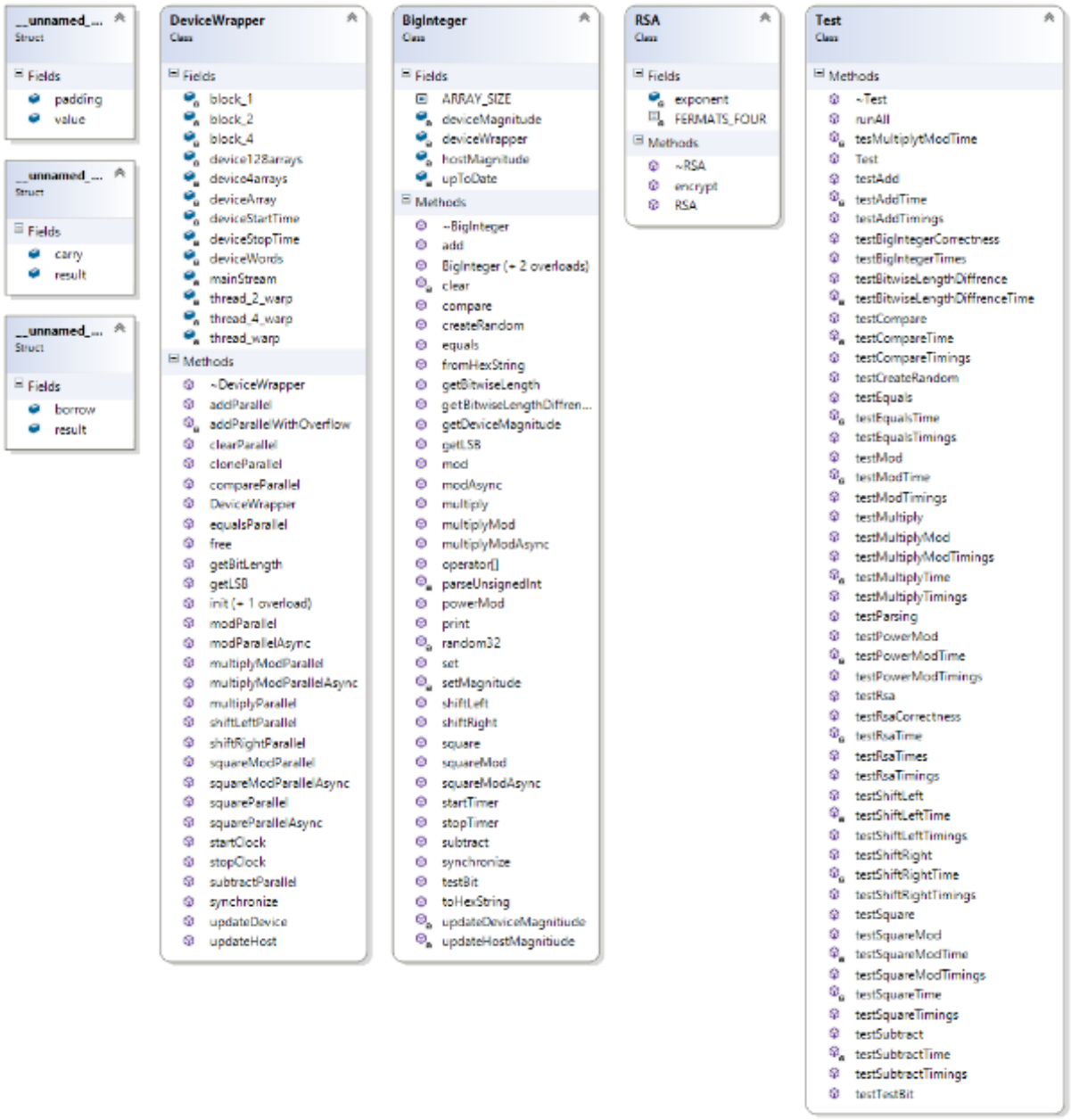

Figure 3.1: Class Diagram

3.1 Parallel equals

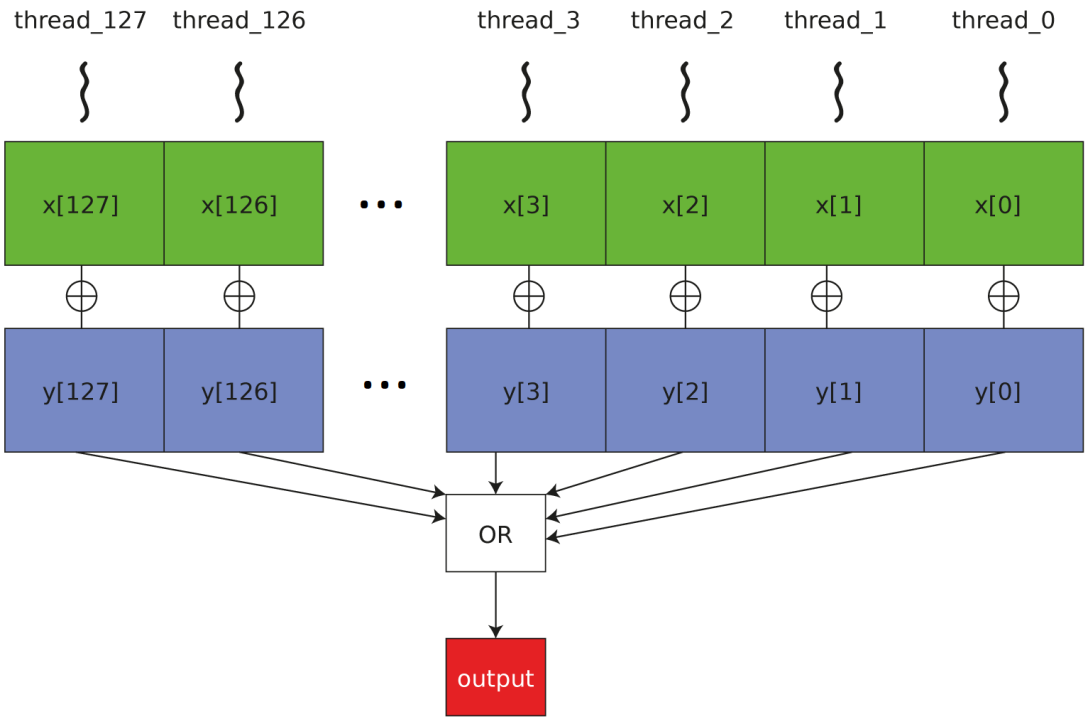

Figure 3.2: Equals function algorithm

The equals algorithm is designed for 128 threads. Each thread "xors" corresponding elements of input arrays. If the values are identical it returns 0. The result are stored in shared memory. Next, all elements of shared array are merged together using bitwise OR, which preserves only set bits. The function returns "true" if the value is 0 after merging every element.

```
shared[0][index] = x[index] ^ y[index];
__syncthreads();
```

3.2 Parallel compare

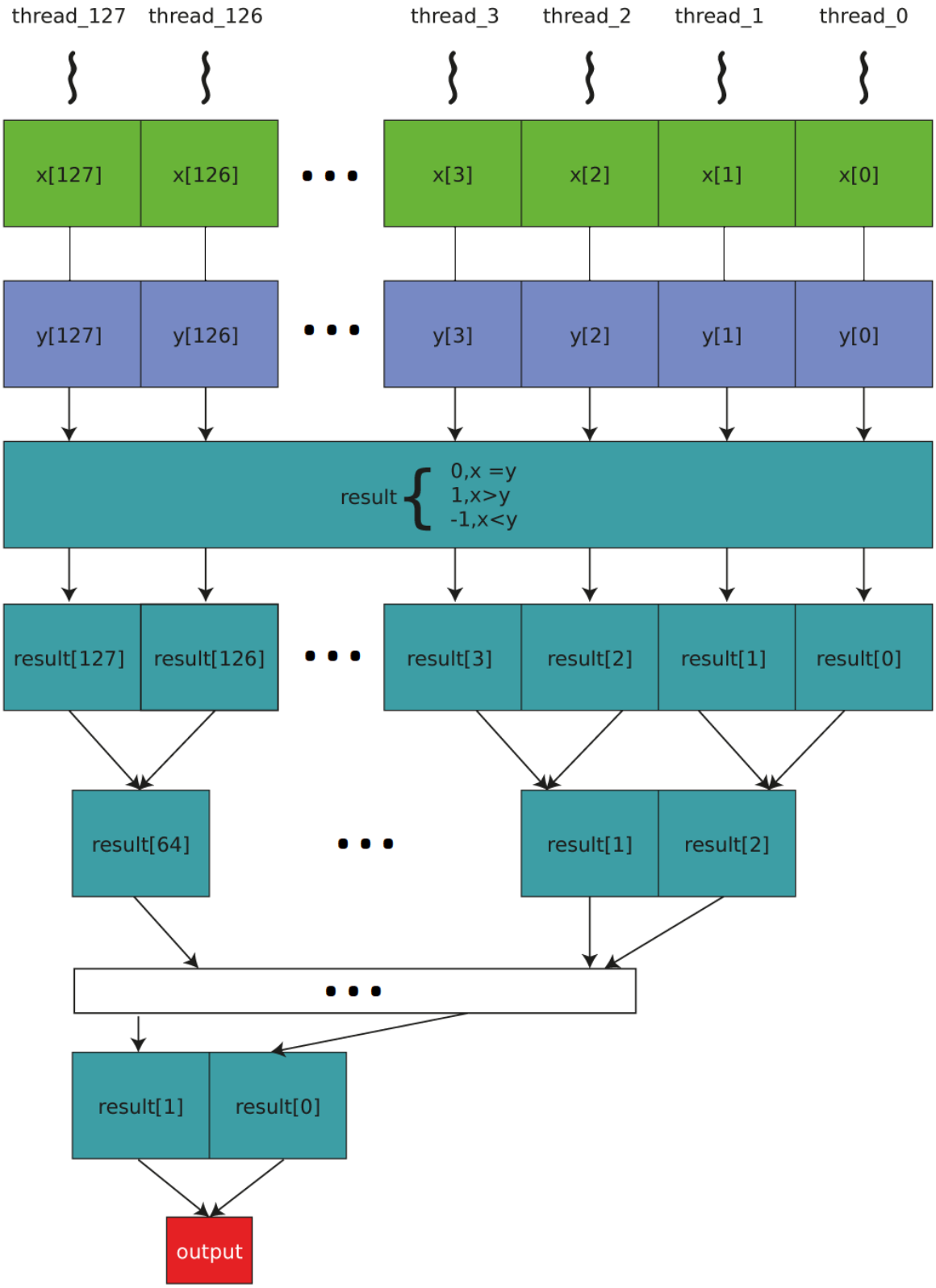

Figure 3.3: Compare function algorithm

The Compare function runs on 128 threads. Each thread separately compares corresponding elements of X and Y. The output is stored in shared memory. Next, the two neighboring elements are merged together.

```
extern "C" __device__ inline int merge(int x, int y)
{
     return x == 0 ? y : x;
}
```

This process needs to be repeated seven times ($log_2 128 = 7$) to reach final result.

```
for (int i = 1; i < 7; i++)
{
     if (1 << (7 - i) > index)     // todo: 6 bank conflicts left
          shared[i][index] = merge(shared[i - 1][globalIndex + 1],
             shared[i - 1][globalIndex]);
     __syncthreads();
}
return merge(shared[6][1], shared[6][0]);
```

3.3 Parallel bit length

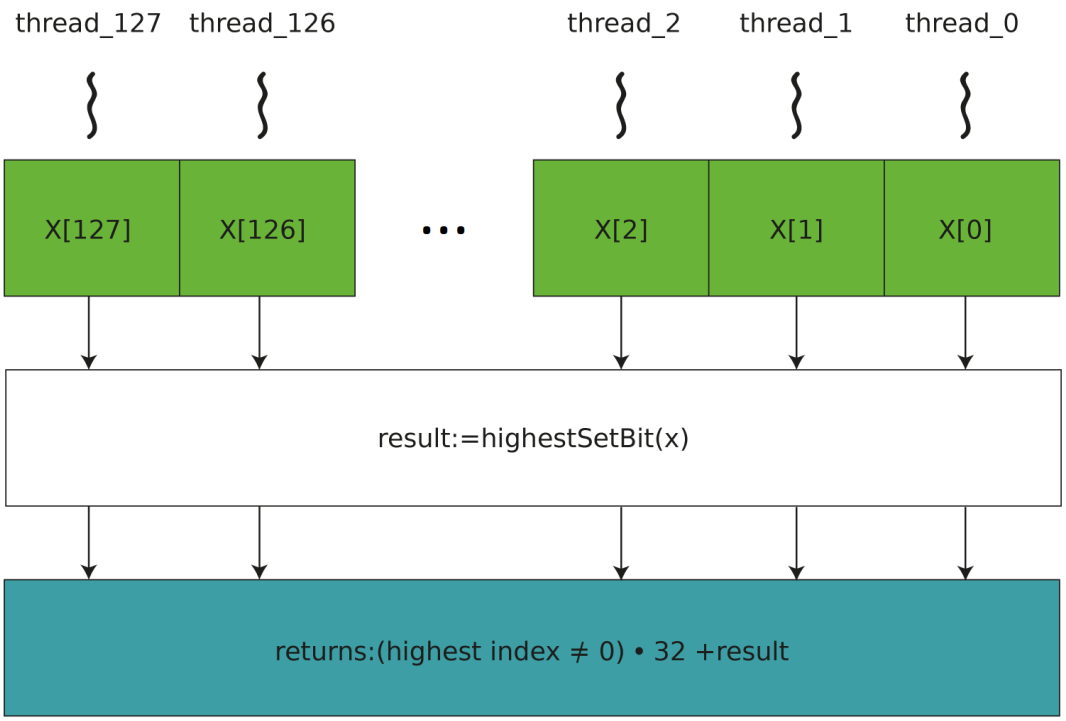

Figure 3.4: Bit length function algorithm

The function returns position of the highest set bit. It runs on 128 threads. Each thread calculates the highest bit for corresponding element in the array, and stores the result in shared memory.

```
#pragma unroll
     for (int i = 0; i < 32; i++)
     {
          if (value >> i == 1)
               bits = i + 1;
     }
     shared[index] = bits;
     __syncthreads();
```

The return value is the index of the highest element multiplied by 32 (bit length of an integer), plus the result of the element.

3.4 Parallel right shift

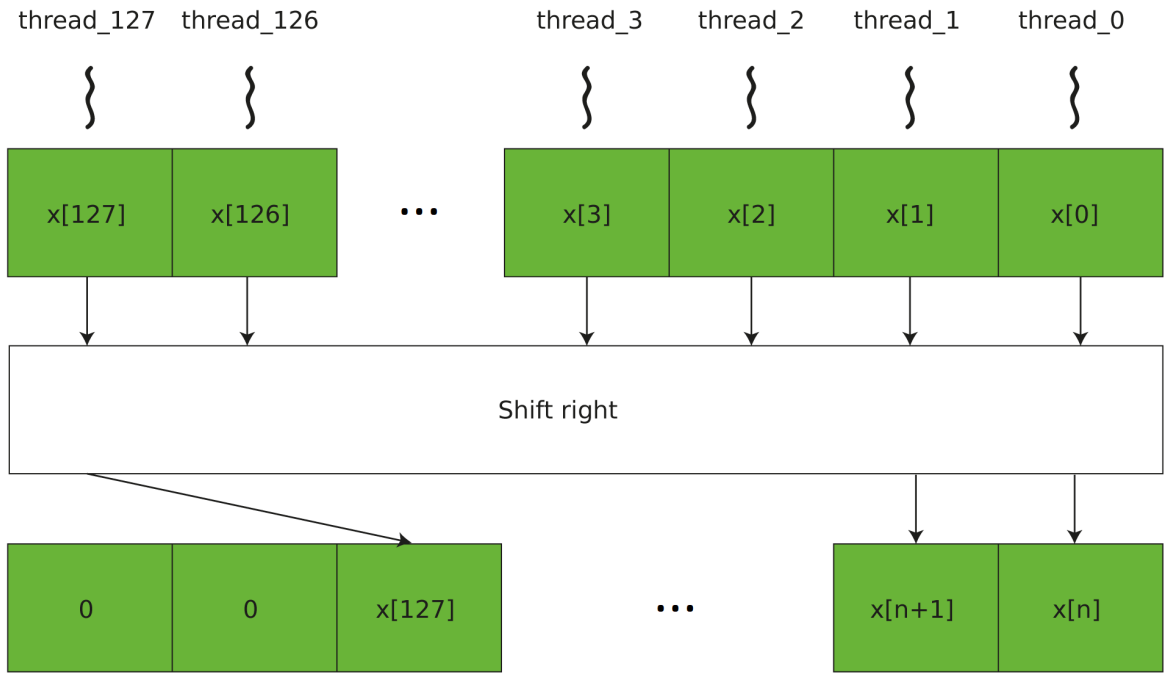

Figure 3.5: Right shift function algorithm

Shift right function runs on 128 threads. Each thread computes the number of elements to move and bits to shift.

```
register int ints = n >> 5;    // n / 32
register int bits = n & 0x1f; // n mod 32
```

First elements of the array are moved and threads are synchronized. Then the bits are shifted.

```
register int remainingBits = 32 - bits;
if (index - 1 >= 0)
     sharedResult[index - 1] = sharedX[index] << remainingBits;
else
```

```
     sharedResult[127] = 0UL;
__syncthreads();
sharedResult[index] = sharedResult[index] | (sharedX[index] >> bits);
__syncthreads();

if (bits > 0)
     x[index] = sharedResult[index];
else
     x[index] = sharedX[index];
```

3.5 Parallel left shift

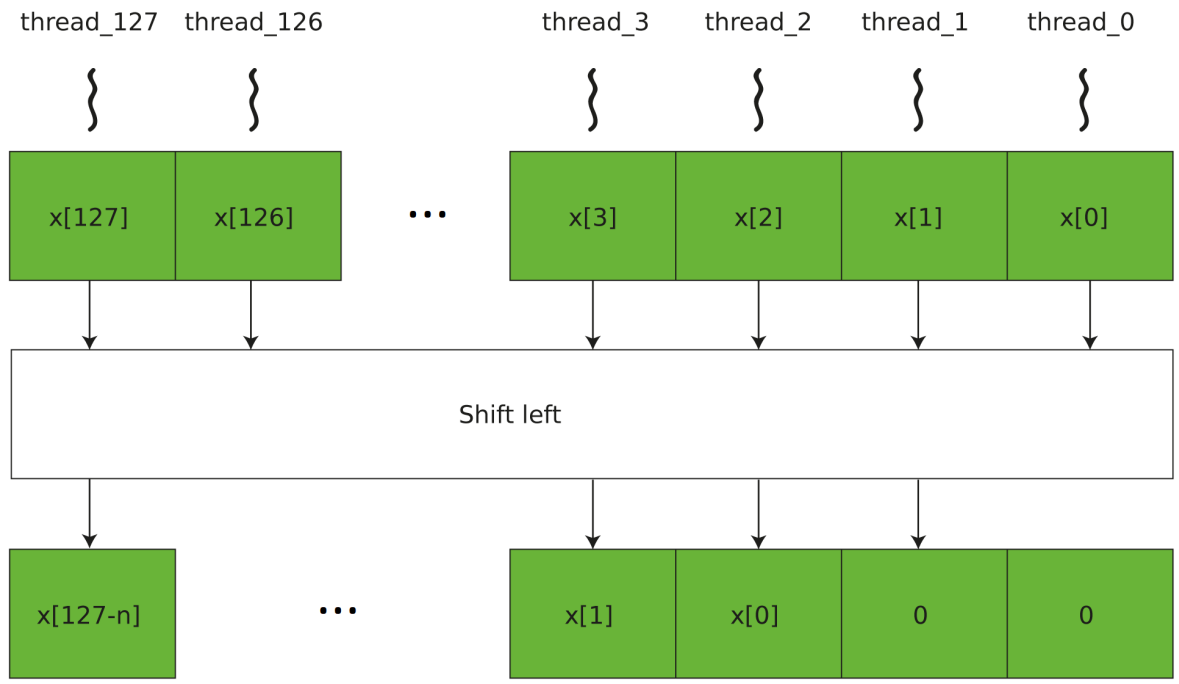

Figure 3.6: Left shift function algorithm

Left shift algorithm is very similar to the right shift. Only difference is inversed direction.

3.6 Parallel add

The add algorithm is designed for 32 threads, which means each thread handles 4 elements of the input array.

```
asm volatile (
     "addc.cc.u32 %0, %1, %2; \n\t"     // propagate and genarate carry
     : "=r"(shared[resultIndex].result[index].value)
     : "r"(x[startIndex + index]), "r"(y[startIndex + index]) : "memory"
        );
```

The carry out from last addition is stored in shared memory.

```
// last iteration generates and stores carry in the array
asm volatile (
     "addc.cc.u32 %0, %2, %3; \n\t"
     "addc.u32 %1, 0, 0; \n\t"
     : "=r"(shared[resultIndex].result[index].value), "=r"(shared[
        resultIndex + 1].carry)
     : "r"(x[startIndex + index]), "r"(y[startIndex + index]) : "memory"
        );

__syncthreads();
```

Next, the stored carries need to be propagated.

```
// first iteration propagates carry from array
asm volatile (
     "add.cc.u32 %0, %0, %1; \n\t" //
     : "+r"(shared[resultIndex].result[index].value)
     : "r"(carry) : "memory");

#pragma unroll
for (index = 1; index < ADDITION_CELLS_PER_THREAD - 1; index++)
{
     asm volatile (
     "addc.cc.u32 %0, %0, 0; \n\t" //propagate generated carries
     : "+r"(shared[resultIndex].result[index].value) :: "memory");
}

// last iteration generates and stores carry in the array
asm volatile (
     "addc.cc.u32 %0, %0, 0; \n\t"
     "addc.u32 %1, 0, 0; \n\t"
     : "+r"(shared[resultIndex].result[index].value), "=r"(shared[
        resultIndex + 1].carry) :: "memory");

__syncthreads();
```

Normally this process would need to be repeated while the carries' array is greater than 0. Since this implementation must be timing attack resistant, it always runs worst cases scenario - 32 iterations.

3.7 Parallel subtract

Implementation of the subtraction is very similar to addition. The only signifying difference is storing the borrow in memory, as it required a simple trick to obtain "1" on the borrow out.

```
// last iteration generates and stores borrow in the array
asm volatile (
     "subc.cc.u32 %0, %2, %3; \n\t"
     "subc.u32 %1, 1, 0; \n\t"     // if borrow out than %1 has 0
        (1-0-1=0), else %1 has 1 (1-0-0=1)
     "xor.b32 %1, %1, 1; \n\t"     // invert 1-->0 and 0-->1
     : "=r"(shared[resultIndex].result[index].value), "+r"(shared[
        resultIndex + 1].borrow)
```

```
    : "r"(x[startIndex + index]), "r"(y[startIndex + index]) : "memory"
      );
```

3.8 Parallel multiply

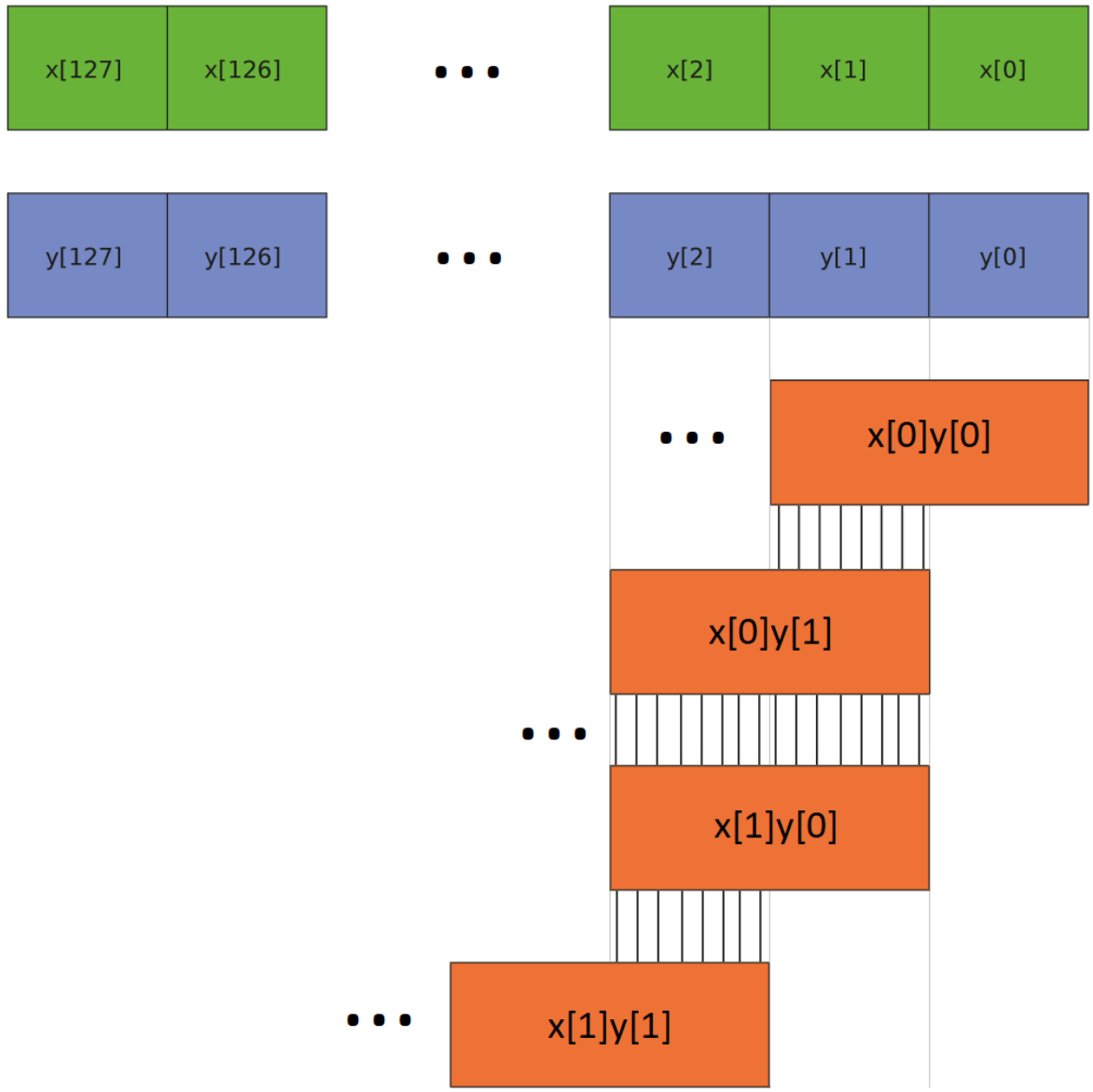

Figure 3.7: Text-book multiplication algorithm

Using classic text-book multiplication algorithm is not the best solution when trying to parallelize, as shown on figuer 3.7. The vertical lines show conflicts in reading and writing to/from shared memory.

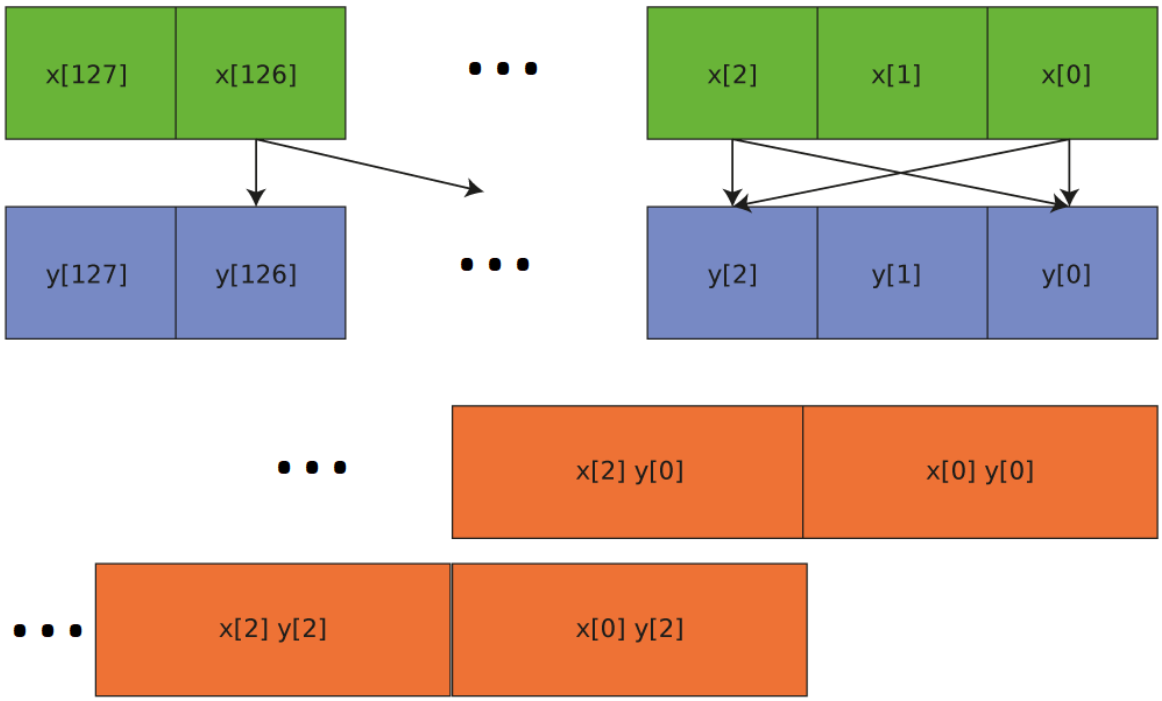

Figure 3.8: Even/Odd multiplication algorithm

The proposed algorithm is to compute intermediate result arrays by multiplying even-even, even-odd, odd-even and odd-odd indices of x and y separately. Two-element offset eliminates conflicts and makes it much easier to synchronize and handle carries. All four arrays can also be computed fully in parallel. Next, the reduction process sums all arrays up using Add function.

```
// parallel multiplication
device_multiply_partial_4 << <block_4, thread_2_warp, 0, mainStream>> >
    (device4arrays, device_x, device_y);

// reduction
device_add_partial_aligned << <block_2, thread_warp, 0, mainStream >> >
    (device4arrays, device4arrays + 2 * 128);

// reduction
device_add_partial_1 << <block_1, thread_warp, 0, mainStream >> > (
    device4arrays, device4arrays + 128);

// set x := result
device_clone_partial_1 << <block_1, thread_4_warp, 0, mainStream >> > (
    device_x, device4arrays);
```

3.9 Parallel modulo reduction

The algorithm must solve:

$$x \bmod m, x \geq m$$

According to CUDA specification[5]: "Integer division and modulo operation are costly as they compiler to up to 20 instructions. They can be replaced with bitwise operations in some cases: If n is a power of 2, (i/n) is equivalent to (i»log2(n)) and (i%n) is equivalent to (i&(n-1))" One way to solve this without any division could be by simply subtracting m from x, until $x \leq m$. However, it would result in tremendous amount of computation, especially if x is significantly larger than m. The following property becomes helpful:

$$x \bmod m = (x \bmod km) \bmod m$$

It basically means that some multiple of m can be used to "pre-reduce" x. Of course x must still be reduced by standalone m, but the value of x is much lower then. Using powers od 2 as factor k, it simplifies to left and right bitwise shifts. The proposed algorithm is to left shift m by the difference in bitwise lengths between x and m. If $m < x$ simply subtract m from x. In other case ($m > x$) right shift m by one bit.

Algorithm 3 X mod M

```
1: xLength ← bitwise length of X
2: mLength ← bitwise length of M
3: bitwiseDifference := xLength - mLength
4: M := M << bitwiseDifference
5: while bitwiseDifference ≥ 0 do
6:     if X > M then
7:         X := X - M
8:     else
9:         M := M >> 1
10: return X
```

The implementation:

```
int xLength = device_getbitlength_partial(x);
int mLength = device_getbitlength_partial(modulus);
int bitwiseDifference = xLength - mLength;
device_shift_left_partial(modulus, bitwiseDifference);
while (bitwiseDifference >= 0)
{
     compare = device_compare_partial(x, modulus);
     if (compare == 1)   // x > m
     {
          device_subtract_partial(x, modulus);
     }
     else if (compare == 0)  // x == m
     s{
          device_clear_partial(x);
          return;
     }
     else // x < m
          {
          device_shift_right_partial(modulus, 1);
          bitwiseDifference--;
     }
     __syncthreads();
}
```

3.10 Parallel multiply modulo

Multiplication is executed using even/odd algorithm without any changes. Modular reduction is injected in after the arrays were generated, and also after every reduction process (summing up).

```
// reduce mod first
device_reduce_modulo_partial_2 << <block_2, thread_4_warp, 0, mainStream
    >> > (device_x, deviceArray, device_m);

// parallel multiplication
device_multiply_partial_4 << <block_4, thread_2_warp, 0, mainStream >> >
    (device4arrays, device_x, deviceArray);

// modular reduction of part-results
device_reduce_modulo_partial_aligned << <block_4, thread_4_warp, 0,
   mainStream >> > (device4arrays, device_m);

// reduction
device_add_partial_aligned << <block_2, thread_warp, 0, mainStream >> >
   (device4arrays, device4arrays + 256);

// modular reduction
device_reduce_modulo_partial_2 << <block_2, thread_4_warp, 0, mainStream
    >> > (device4arrays, device4arrays + 128, device_m);

// reduction
device_add_partial_1 << <block_1, thread_warp, 0, mainStream >> > (
   device4arrays, device4arrays + 128);

// final modular reduction
device_reduce_modulo_partial_1 << < block_1, thread_4_warp, 0,
   mainStream >> > (device4arrays, device_m);

// set x := result
device_clone_partial_1 << <block_1, thread_4_warp, 0, mainStream >> > (
   device_x, device4arrays);
```

3.11 Parallel power modulo

This function uses the Montgomery Ladder algorithm. Implementations is rather simple.

```
BigInteger x0(1);
BigInteger x1(*this);

for (int bits = exponent.getBitwiseLength() - 1; bits >= 0; bits--)
{
     if (exponent.testBit(bits))
     {
          x0.multiplyModAsync(x1, modulus);
          x1.multiplyModAsync(x1, modulus);
          }
     else
     {
          x1.multiplyModAsync(x0, modulus);
```

```
            x0.multiplyModAsync(x0, modulus);
        }
        x0.synchronize();
        x1.synchronize();
}
set(x0);
```

The multiplication of x0 and x1 are executed in parallel.

Chapter 4

Testing

This chapter presents the behavior of implemented algorithms based on the inputs' lengths. All of the functions are designed to handle data up to 4096 bits long. Each test is repeated 100-300 times (depends on the complexity) to provide rather average results.

Other test were executed using CUDA Profiler Activity. "This tool gathers detailed performance information, in addition to timing and launch configuration details. A CUDA Profiler activity consists of a kernel filter and a set of profiler experiments. Profile experiments are directed analysis tests targeted at collecting in-depth performance information for an isolated instance of a kernel launch."[7]

4.1 CUDA experiments

4.1.1 Occupancy

"A warp (32 threads) is considered active from the time its threads begin executing to the time when all threads in the warp have exited from the kernel. There is a maximum number of warps which can be concurrently active on a Streaming Multiprocessor (SM). Occupancy is defined as the ratio of active warps on an SM to the maximum number of active warps supported by the SM. Occupancy varies over time as warps begin and end, and can be different for each SM.

Low occupancy results in poor instruction issue efficiency, because there are not enough eligible warps to hide latency between dependent instructions. When occupancy is at a sufficient level to hide latency, increasing it further may degrade performance due to the reduction in resources per thread.

4.1.2 Theoretical occupancy

There is an upper limit for active warps, and thus also for occupancy, derivable from the launch configuration, compile options for the kernel, and device capabilities. Each block of a kernel launch gets distributed to one of the SMs for execution. A block is considered active from the time its warps begin executing to the time when all warps in the block have exited from the kernel. The number of blocks which can execute concurrently on an SM is limited by the factors listed below. The upper limit for active warps is the product of the upper limit for active blocks and the number of warps per block. Thus, the upper limit for active warps can be raised by increasing the number of warps per block (defined

by block dimensions), or by changing the factors limiting how many blocks can fit on an SM to allow more active blocks. The limiting factors are:

- Warps per SM

  The SM has a maximum number of warps that can be active at once. Since occupancy is the ratio of active warps to maximum supported active warps, occupancy is 100% if the number of active warps equals the maximum. If this factor is limiting active blocks, occupancy cannot be increased. For example, on a GPU that supports 64 active warps per SM, 8 active blocks with 256 threads per block (8 warps per block) results in 64 active warps, and 100% theoretical occupancy. Similarly, 16 active blocks with 128 threads per block (4 warps per block) would also result in 64 active warps, and 100% theoretical occupancy.

- Blocks per SM

  The SM has a maximum number of blocks that can be active at once. If occupancy is below 100% and this factor is limiting active blocks, it means each block does not contain enough warps to reach 100% occupancy when the device's active block limit is reached. Occupancy can be increased by increasing block size. For example, on a GPU that supports 16 active blocks and 64 active warps per SM, blocks with 32 threads (1 warp per block) result in at most 16 active warps (25% theoretical occupancy), because only 16 blocks can be active, and each block has only one warp. On this GPU, increasing block size to 4 warps per block makes it possible to achieve 100% theoretical occupancy.

- Registers per SM

  The SM has a set of registers shared by all active threads. If this factor is limiting active blocks, it means the number of registers per thread allocated by the compiler can be reduced to increase occupancy. Kernel execution time and average eligible warps should be monitored carefully when adjusting registers per thread to control occupancy. The performance gain from improved latency hiding due to increased occupancy may be outweighed by the performance loss of having fewer registers per thread, and spilling to local memory more often. The best-performing balance of occupancy and registers per thread can be found experimentally by tracing the kernel compiled with different numbers of registers per thread.

- Shared memory per SM

  The SM has a fixed amount of shared memory shared by all active threads. If this factor is limiting active blocks, it means the shared memory needed per thread can be reduced to increase occupancy. Shared memory per thread is the sum of static shared memory, the total size needed for all '__shared__' variables, and dynamic shared memory, the amount of shared memory specified as a parameter to the kernel launch. For some CUDA devices, the amount of shared memory per SM is configurable, trading between shared memory size and L1 cache size. If such a GPU is configured to use more L1 cache and shared memory is the limiting factor for occupancy, then occupancy can also be increased by choosing to use less L1 cache and more shared memory.

4.1.3 Achieved occupancy

Theoretical occupancy shows the upper bound active warps on an SM, but the true number of active warps varies over the duration of the kernel, as warps begin and end. An SM contain one or more warp schedulers. Each warp scheduler attempts to issue instructions from a warp on each clock cycle. To sufficiently hide latencies between dependent instructions, each scheduler must have at least one warp eligible to issue an instruction every clock cycle. Maintaining as many active warps as possible (a high occupancy) throughout the execution of the kernel helps to avoid situations where all warps are stalled and no instructions are issued. Achieved occupancy is measured on each warp scheduler using hardware performance counters to count the number of active warps on that scheduler every clock cycle. These counts are then summed across all warp schedulers on each SM and divided by the clock cycles the SM is active to find the average active warps per SM. Dividing by the SM's maximum supported number of active warps gives the achieved occupancy per SM averaged over the duration of the kernel. Averaging across all SMs gives the overall achieved occupancy.

4.1.4 Occupancy charts

- Varying Block Size

  Shows how varying the block size while holding other parameters constant would affect the theoretical occupancy. The circled point shows the current number of threads per block and the current upper limit of active warps. If the chart's line goes higher than the circle, changing the block size could increase occupancy without changing the other factors.

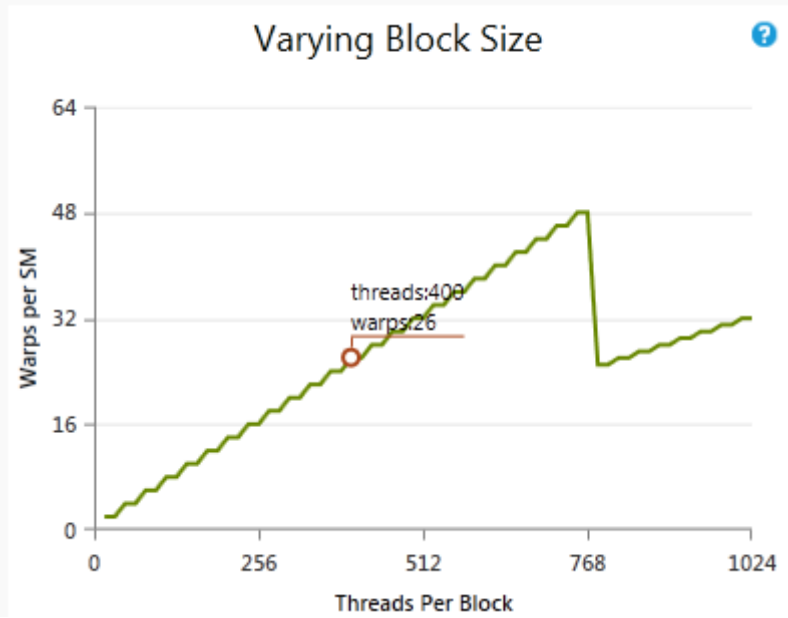

Figure 4.1: Varying block size chart example

- Varying Register Count

  Shows how varying the register count while holding other parameters constant would affect the theoretical occupancy. The circled point shows the current number of registers per thread and the current upper limit of active warps. If the chart's

line goes higher than the circle, changing the number of registers per thread could increase occupancy without changing the other factors.

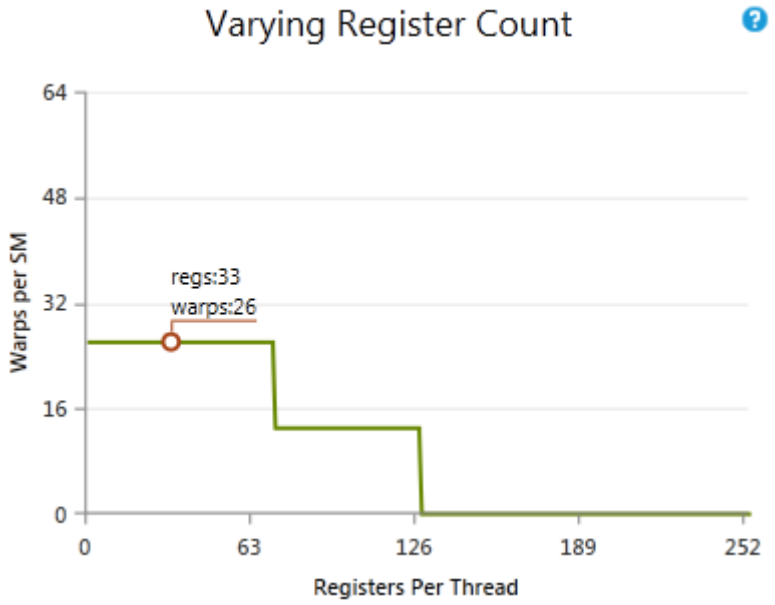

Figure 4.2: Varying register count chart example

- Varying Shared Memory Usage

  Shows how varying the shared memory usage while holding other parameters constant would affect the theoretical occupancy. The circled point shows the current amount of shared memory per block and the current upper limit of active warps. If the chart's line goes higher than the circle, changing the amount of shared memory per block could increase occupancy without changing the other factors.

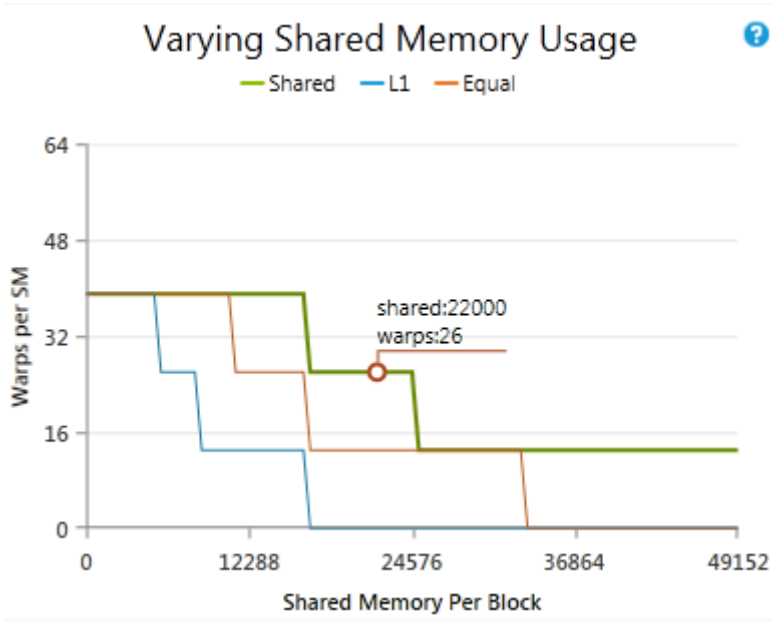

Figure 4.3: Varying shared memory usage chart example

- Achieved Occupancy Per SM

The achieved occupancy for each SM. The values reported are the average across all warp schedulers for the duration of the kernel execution. The line across all bars is the average, which is the number reported as Achieved Occupancy in the other tables.

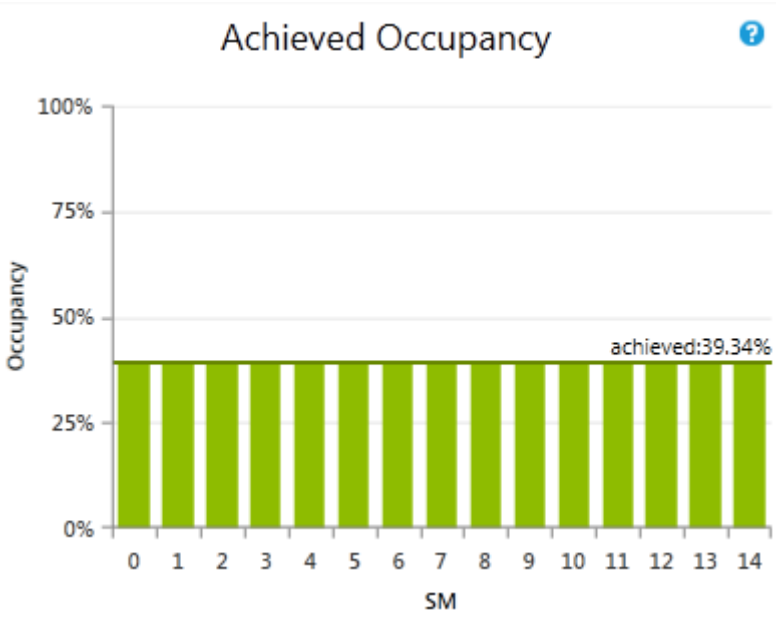

Figure 4.4: Achieved o ccupancy per SM chart example

4.1.5 Instruction statistics

The Instruction statistics experiment provides a first level triage for understanding the overall utilization of the target device when executing the kernel.

The global work distribution engine schedules thread blocks to Streaming Multiprocessors. A multiprocessor is considered to be active if it has at least one warp assigned to it. The total number of cycles a SM was active for the duration of the kernel execution is defined as Active Cycles.

Each multiprocessor exposes multiple warp schedulers that are able to execute at least one instruction per cycle. At every instruction issue time, each warp scheduler selects one warp that is able to make forward process from its assigned list of warps. For this selected warp the scheduler then issues either the next single instruction or the next two instructions.

A warp scheduler might need to issue an instruction multiple times to actually complete the execution for all 32 threads of a warp. The two primary reasons for this difference between Instructions Issued and Instructions Executed are: First, address divergence and bank conflicts on memory operations. Second, assembly instructions that can only be issued for a half-warp per cycle and thus need to be issued twice. Double floating-point instructions are the prime example for such instructions. As each executed instruction needs to be at least issued once, the following statement holds true in all cases:

$$\text{Instructions Issued} >= \text{Instructions Executed}$$

Issuing an instruction multiple times is also referred to as Instruction Replay. Each replay iteration takes away the ability to make forward progress by issuing new instructions

on that warp scheduler. Also the compute resources required to process the instruction are consumed for every instruction replay. In short, the more instruction replay iterations are required the higher is the performance impact on the kernel execution.

- Instructions Per Clock (IPC)

  A z-ordered column graph showing the achieved instructions throughputs per SM for both, issued instructions and executed instructions. The theoretical maximum peak IPC is a device limit and defined by the compute capabilities of the target device. The y-axis is scaled to this peak value.

  - Issued IPC

    The average number of issued instructions per cycle accounting for every iteration of instruction replays. Optimal if as close as possible to the Executed IPC. Some assembly instructions require to be multi-issued, hence the instruction mix affects the definition of the optimal target for this metric.

  - Executed IPC

    The average number of executed instructions per cycle. Higher numbers indicate more efficient usage of the available all resources. As each warp scheduler of a multiprocessor can execute instructions independently, a target goal of executing one instruction per cycle means executing on average with an IPC equal to the number of warp schedulers per SM. The maximum achievable target IPC for a kernel is dependent on the mixture of instructions executed.

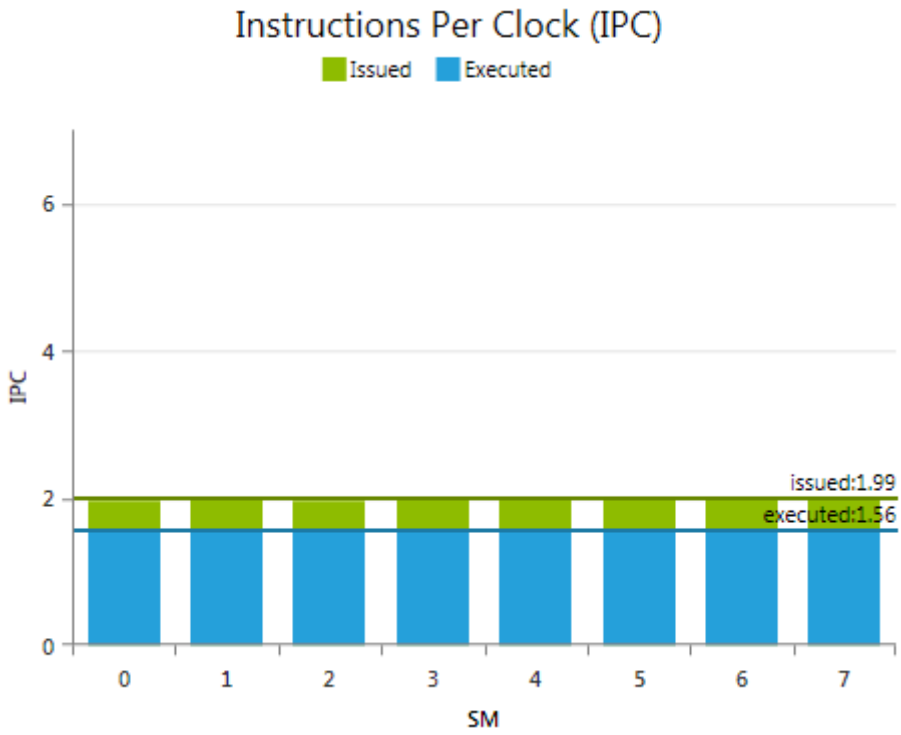

Figure 4.5: Instructions per clocks chart example

- SM Activity Shows the percentage of time each multiprocessor was active during the duration of the kernel launch. A multiprocessor is considered to be active if at least one warp is currently assigned for execution. An SM can be inactive - even

though the kernel grid is not yet completed - due to high workload imbalances. Such uneven balancing between the SMs can be caused by a few factors: Different execution times for the kernel blocks, variations between the number of scheduled blocks per SM, or a combination of the two.

The observable result of a load imbalance are highly different activity values across the multiprocessors; simply caused by the fact that some SMs are still busy executing work, while others SMs already completed their share of work and stay idle as no more work items are left to be scheduled. This is typically referred to as a "tail effect". Small kernel grids with a low number of blocks are more likely to be affected by a tail effects.

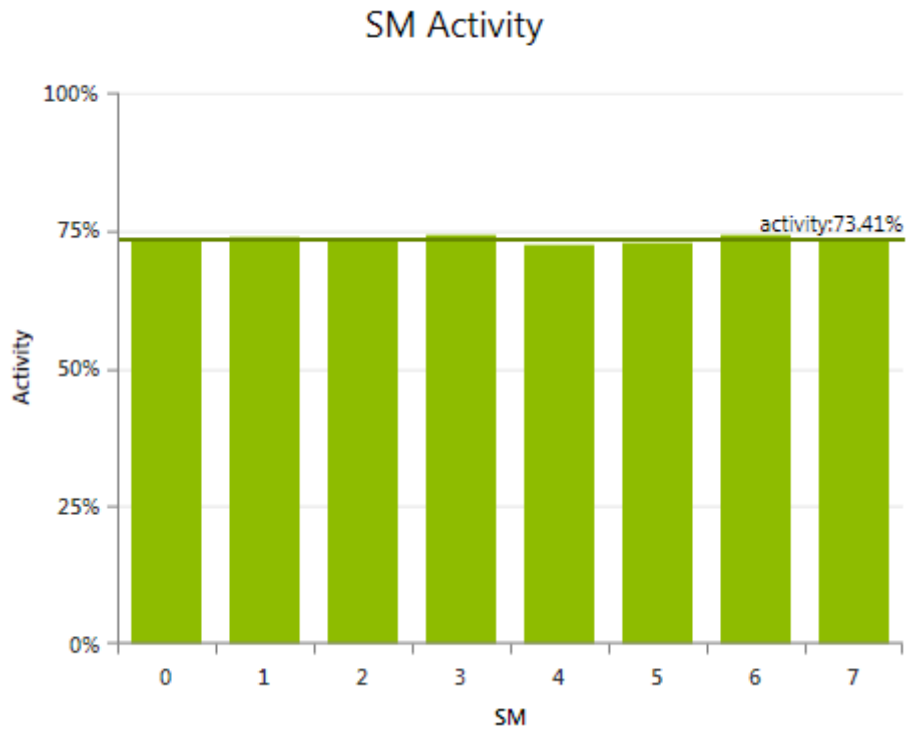

Figure 4.6: SM activity chart example

- Instructions Per Warp (IPW) Shows the average executed instructions per warp for each multiprocessor. High variations in the IPW metric across the SMs indicate non-uniform workloads for the blocks of the kernel grid. While such imbalance does not necessarily have to result in low performance, IPW is very useful to understand the cause of variations in SM Activity.

  The most common code pattern to cause high variations in IPW is conditionally executed code blocks where the conditional expression is dependent on the block index. Examples include: special pre-processing or post-processing operations executed for a single block only, or costly detection and handling of edge conditions that are only triggered for some subset of the grid.

- Warps Launched Shows the total number of warps launched per multiprocessor for the executed kernel grid. Large differences in the number of warps launched per SM are most commonly the result of providing an insufficient amount of parallelism with the kernel grid. More specifically, the number of kernel blocks is too low to make good use of all available compute resources. A high variation in the number

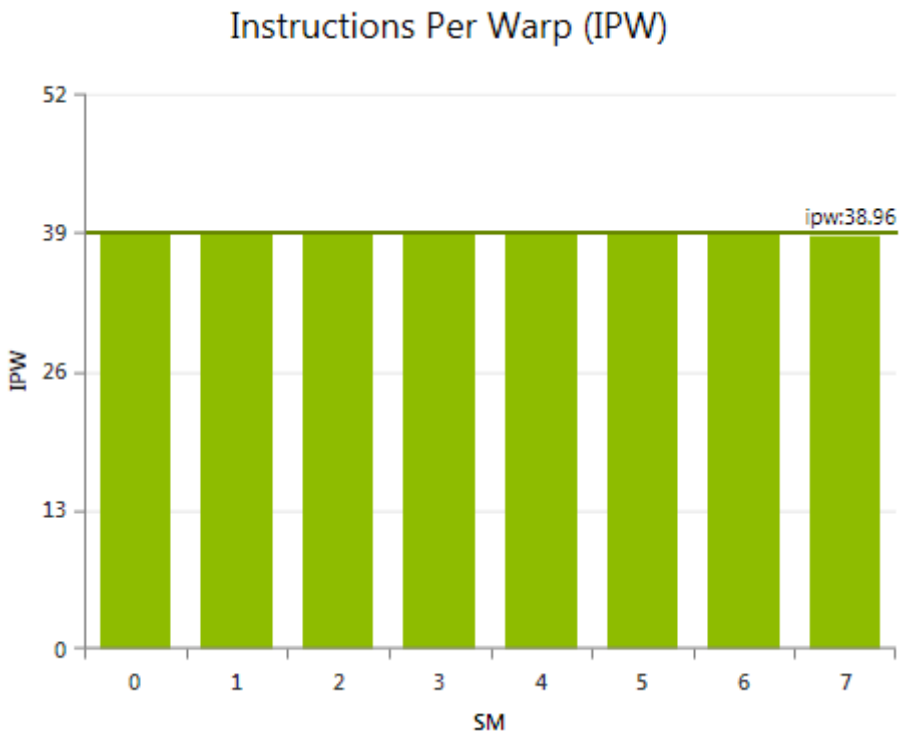

Figure 4.7: Instructions per warp chart example

of warps launched is only a concern if the SM Activity is low on one or more SMs. In this case, partitioning workload that either result in less variance in execution duration per warp or in the execution of more thread blocks. Both cases will help the work distributor in dispatching the given work more evenly.

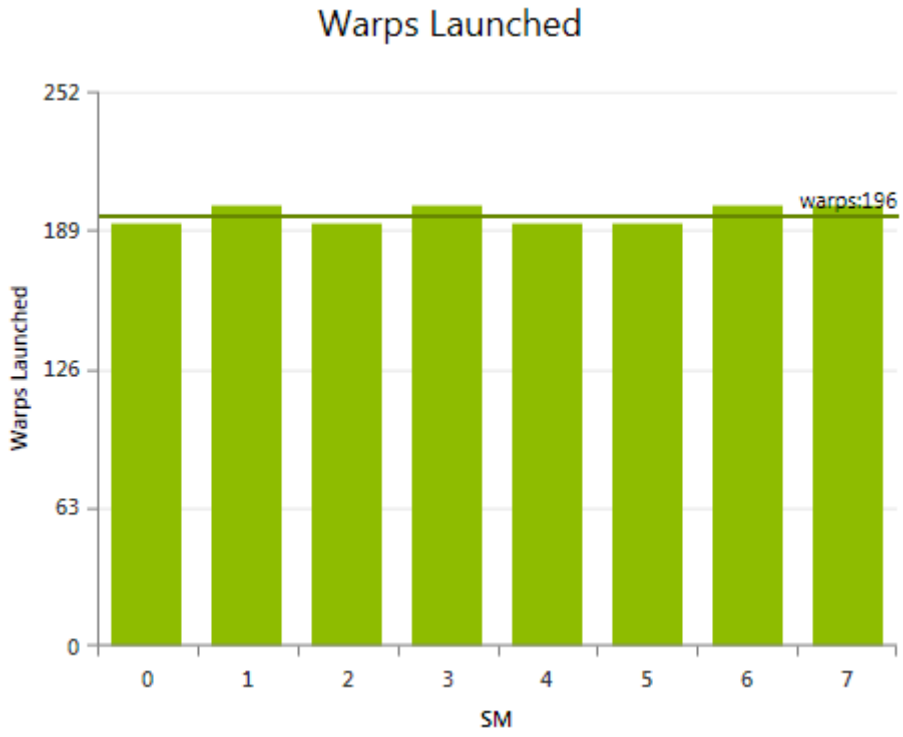

Figure 4.8: Warps launched chart example

4.1.6 Branch statistics

Flow control can have a serious impact on the efficiency of executing a kernel. Especially if a lot of flow control decisions are divergent, forcing the threads of a warp into very different control paths through the kernel code. The Branch Statistics experiment helps answering the question of how often flow control instructions were executed, how many of them were uniform versus divergent, and how much the flow control impacted the overall kernel execution performance. A flow control instruction is considered to be divergent if it forces the threads of a warp to execute different execution paths. If this happens, the different execution paths must be serialized, since all of the threads of a warp share a program counter; this increases the total number of instructions executed for this warp. When all the different execution paths have completed, the threads converge back to the same execution path. Conditional expressions that evaluate to a uniform decision across all threads of a warp, do only execute the single, selected code path - consequently causing a lot less overhead. The ratio of executed uniform flow control decisions over all executed conditionals is defined as Branch Efficiency.

The actual performance impact caused by divergent flow control is proportional to the combination of how many different code paths need to be evaluated and how expensive the serialized code segments are. One way of capturing this is to track for all executed instructions how many of the threads in a warp were actually participating in the execution, i.e. how many threads were not predicated off. This is typically referred to as Control Flow Efficiency. By definition this is independent of the number of flow control decisions made, but rather states an upper limit of the utilization of the available compute resources due to flow control.

- Efficiency

  The efficiency chart shows the two primary metrics for evaluating the impact of flow control.

  - Branch Efficiency

    States the ratio of uniform control flow decisions over all executed branch instructions. Shown per-SM (the bars) and averaged over all SMs (the Branch line). Higher values are better, as warps more often take a uniform code path. A value lower than 100% is a necessary, but not sufficient indicator for a negative impact on the kernel execution performance, since the metric does not have any knowledge about the size of the code regions enclosed by the conditionals. For example, one divergent flow control decision out of ten executed branches may be negligible if it encloses very few lines of code only; but it may have a huge impact if it forced the warp to execute many different code paths with thousands of instructions.

  - Control Flow Efficiency

    Defined as the ratio of active threads that are not predicated off over the maximum number of threads per warp for each executed instruction. Gets lower with fewer threads per warp being active per instruction; therefore serving as a metric for the efficiency in using the available processing units. Lower control flow efficiency can be caused by: Launching warps with less than 32 threads active. Terminating some threads in a warp earlier than others. Or executing instructions with only a subset of the threads enabled.

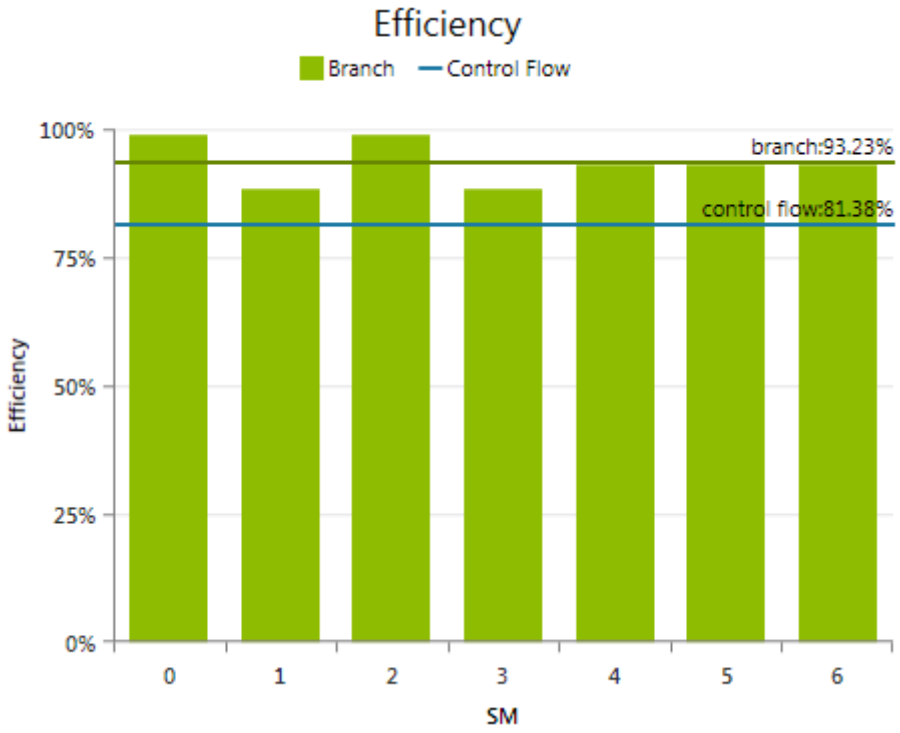

Figure 4.9: Efficiency chart example

- Branches per warp

  Shows the average count of executed branch instructions per warp per SM grouped by the outcome of the evaluation of the conditional statement. Useful to investigate the total amount of flow control instructions executed for the warps of the kernel grid.

  - Not Taken / Taken

    Average number of executed branch instructions with a uniform control flow decision per warp; that is all active threads of a warp either take or not take the branch

  - Diverged

    Average number of executed branch instruction per warp for which the conditional resulted in different outcomes across the threads of the warp. All code paths with at least one participating thread get executed sequentially. Lower numbers are better.

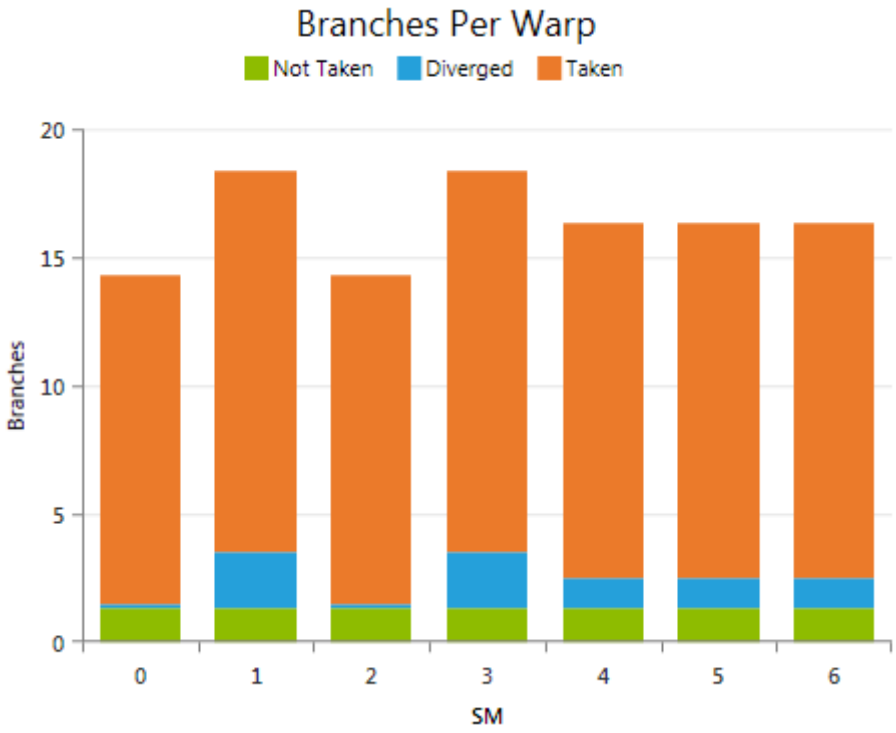

Figure 4.10: Branches per warp chart example

- Branches condition

  Shows the distribution of executed branches that were uniform versus divergent aggregated across all warps of the kernel grid.

  - Not Taken / Taken

    Total number of executed branch instructions with a uniform control flow decision; that is all active threads of a warp either take or not take the branch.

  - Diverged

    Total number of executed branch instruction for which the conditional resulted in different outcomes across the threads of the warp. All code paths with at least one participating thread get executed sequentially. Lower numbers are better.

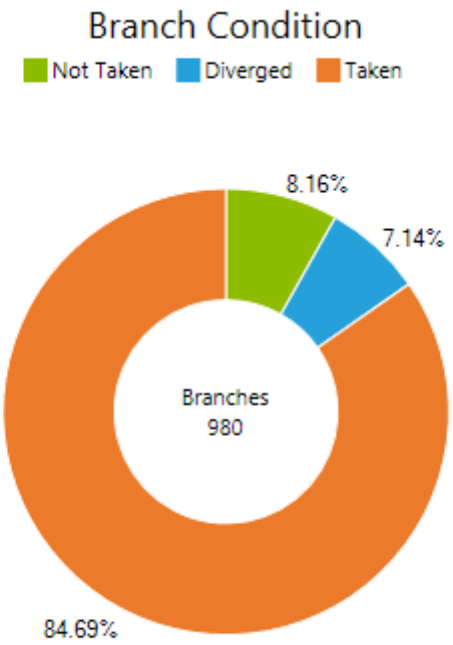

Figure 4.11: Branch condition chart example

4.1.7 Issue efficiency

The issue efficiency experiment provides information about the device's ability to issue the instructions. The key takeaway is the answer to the question if the device was able to issue instructions every cycle. Not being able to do so inevitably lowers the potential peak performance of the kernel.

- Warps Per SM

  The metrics are reported as average values across the complete kernel execution for each individual SM of the target device. The y-Axis is scaled to the device limit.

  - Eligible Warps

    An active warp is considered eligible if it is able to issue the next instruction. Each warp scheduler will select the next warp to issue an instruction from the pool of eligible warps. Warps that are not eligible will report an Issue Stall Reason. The target is to have at least one eligible warp per scheduler per cycle.

  - Theoretical Occupancy

    The theoretical occupancy acts as upper limit to active warps and consequently also eligible warps per SM. It is defined by the execution configuration of the kernel launch.

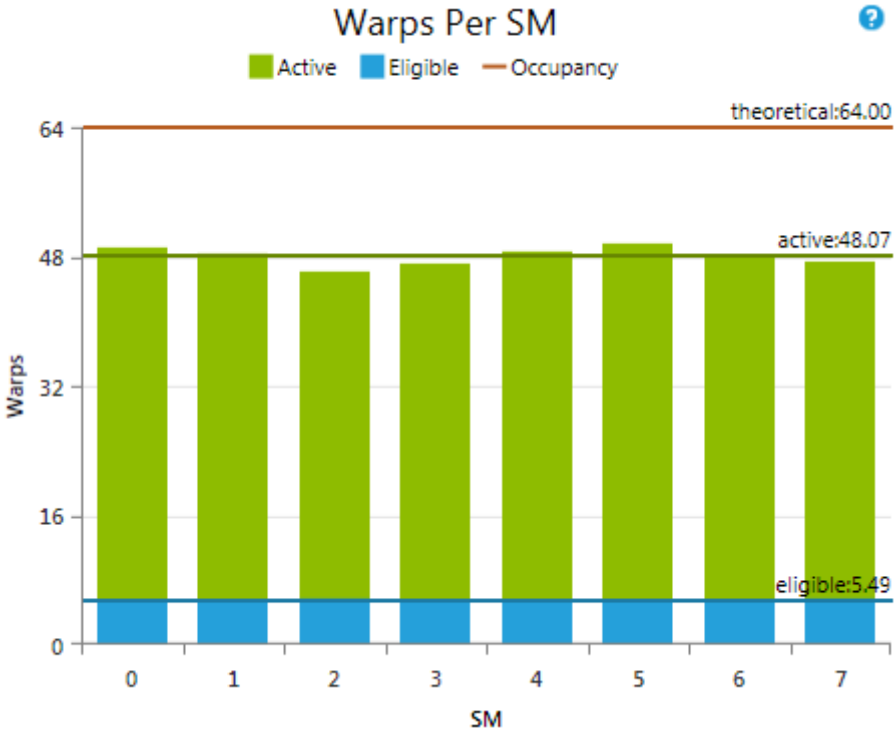

Figure 4.12: Warps per SM chart example

- Warp Issue Efficiency

  On every clock cycle, a warp scheduler tries to issue an instruction from one of its warps. When a warp issues an instruction, it takes at least a few cycles before it becomes eligible to issue again, so many warps should be active on a warp scheduler to ensure it can issue an instruction from some warp on every cycle. In this experiment, the profiler counts whether an instruction was issued or not for each clock cycle on each warp scheduler. The Warp Issue Efficiency chart shows the average across all warp schedulers over the duration of kernel execution.

  – No Eligible

    The number of cycles that a warp scheduler had no eligible warps to select from and therefore did not issue an instruction. The lower the percentage of cycles with no eligible warp the more efficient the code runs on the target device.

  – One or More Eligible

    The number of cycles that a warp scheduler had at least one eligible warps to select from. This metric is equal to total number of cycles an instruction was issued summed across all warp schedulers. Better if the value is higher with a target of getting close to 100%.

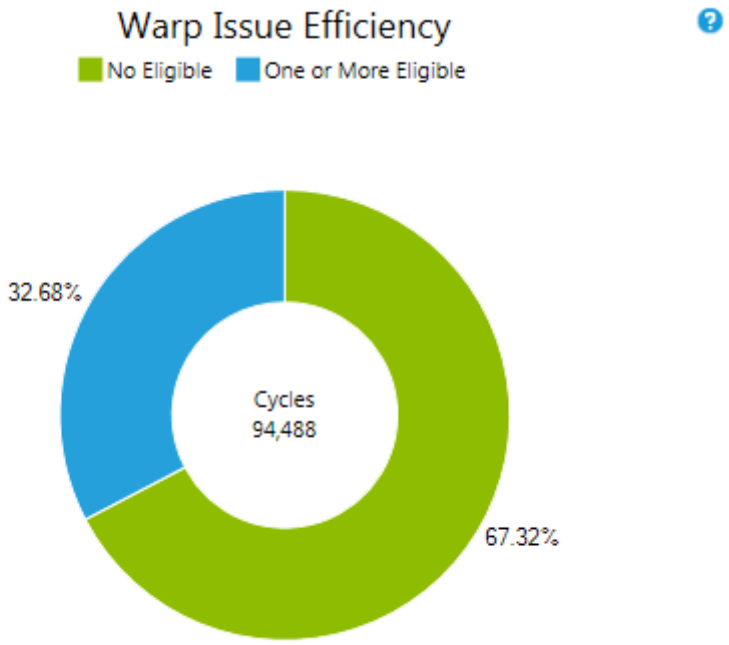

Figure 4.13: Warp Issue Efficiency chart example

- Issue Stall Reasons

  The issue stall reasons capture why an active warp is not eligible. On devices of compute capability 3.0 and higher, every stalled warp increments its most critical stall reason by one on every cycle. The sum of the stall reasons, hence increment per multiprocessor per cycle, by a value between zero (if all warps are eligible) and the number of active warps (if all warps are stalled). The update of the stall reason counters occurs for all stalled warps independent of being able to issue an instruction that cycle or not.

  – Pipeline Busy
    The compute resources required by the instruction are not yet available.
  – Texture
    The texture subsystem is fully utilized, or has too many outstanding requests
  – Constant
    A constant load is blocked due to a miss in the constants cache.
  – Instruction Fetch
    The next assembly instruction has not yet been fetched.
  – Memory Throttle
    A large number of pending memory operations prevent further forward progress. These can be reduced by combining several memory transactions into one.
  – Memory Dependency
    A load/store cannot be made because the required resources are not available or are fully utilized, or too many requests of a given type are outstanding. Memory dependency stalls can potentially be reduced by optimizing memory alignment and access patterns.

– Synchronization
  The warp is blocked at a _syncthreads() call.
– Execution Dependency
  An input required by the instruction is not yet available. Execution dependency stalls can potentially be reduced by increasing instruction-level parallelism.

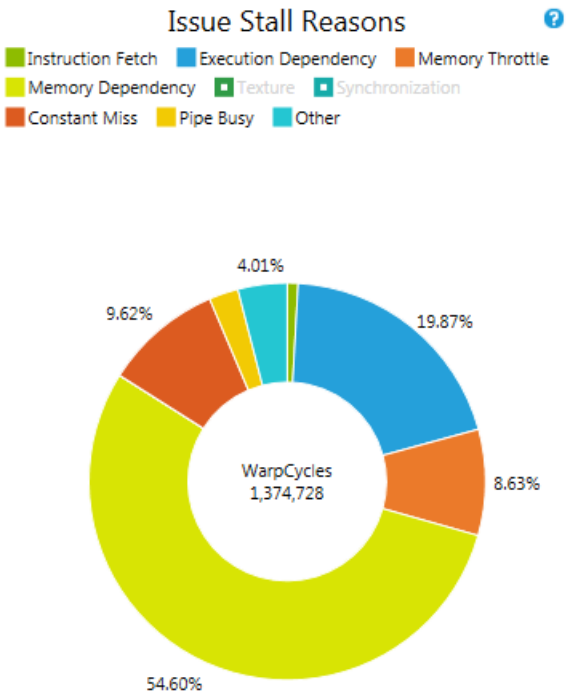

Figure 4.14: Issue Stall Reason chart example

4.1.8 Pipe utilization

Each Streaming Multiprocessor (SM) of a CUDA device features numerous hardware units that are specialized in performing specific task. At the chip level those units provide execution pipelines to which the warp schedulers dispatch instructions to. For example, texture units provide the ability to execute texture fetches and perform texture filtering. Load/Store units fetch and save data to memory. Understanding the utilization of those pipelines and knowing how close they are to the peak performance of the target device are key information for analyzing the efficiency of executing a kernel; and also allows to identify performance bottlenecks caused by oversubscribing to a certain type of pipeline.

Pipeline Utilization metrics report the observed utilization for each pipeline at runtime. High pipeline utilization states that the corresponding compute resources were used heavily and kept busy often during the execution of the kernel. Low values indicate that the pipeline is not frequently used and resources were idle. The results for individual pipelines are independent of each other; summing up two or more pipeline utilization percentages does not result in a meaningful value. As the pipeline metrics are reported as an average over the duration of the kernel launch, a low value does not necessarily rule out that the pipeline was a bottleneck at some point in time during the kernel execution.

- Pipe Utilization Chart

Shows the average utilization of the four major logical pipelines of the SMs during the execution of the kernel. Useful for investigating if a pipeline is oversubscribed and therefore is limiting the kernel's performance. Also helpful to estimate if adding more work will scale well or if a pipeline limit will be hit. In this context adding more work may refer to adding more arithmetic workload (for example by increasing the accuracy of some calculations), increasing the number of memory operations (including introducing register spilling), or increasing the number of active warps per SM with the goal of improving instruction latency hiding.

- Load / Store

  Covers all issued instructions that trigger a request to the memory system of the target device - excluding texture operations. Accounts for load and store operations to global, local, shared memory as well as any atomic operation. Also includes register spills. Devices of compute capability 3.5 and higher support loading global memory through the read-only data cache (LDG); those operations do not contribute to the load/store group, but are accounted for in the texture pipeline utilization instead.

- Texture

  Covers all issued instructions that perform a texture fetch and, for devices of compute capability 3.5 and higher, global memory loads via the read-only data cache.

- Control Flow

  Covers all issued instructions that can have an effect on the control flow, such as branch instructions (BRA,BRX), jump instructions (JMP,JMX), function calls (CAL,JCAL), loop control instructions (BRK,CONT), return instructions (RET), program termination (EXIT), and barrier synchronization (BAR).

- Arithmetic

  Covers all issued floating point instructions, integer instructions, conversion operations, and movement instructions.

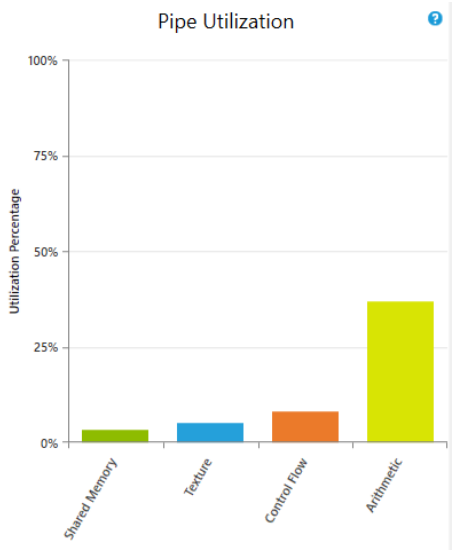

Figure 4.15: Pipe Utilization chart example

- Arithmetic Workload

  Provides the distribution of estimated costs for numerous classes of arithmetic instructions. The cost model is based on the issue count weighted by the reciprocal of the corresponding instruction throughput.

  - FP16

    Estimated workload for all 16-bit floating-point add (HADD), multiply (HMUL), multiply-add (HFMA) instructions.

  - FP32

    Estimated workload for all 32-bit floating-point add (FADD), multiply (FMUL), multiply-add (FMAD) instructions.

  - FP64

    Estimated workload for all 64-bit floating-point add (DADD), multiply (DMUL), multiply-add (DMAD) instructions.

  - FP32 (Special)

    Estimated workload for all 32-bit floating-point reciprocal (RCP), reciprocal square root (RSQ), base-2 logarithm (LG2), base 2 exponential (EX2), sine (SIN), cosine (COS) instructions.

  - I32 (Add)

    Estimated workload for all 32-bit integer add (IADD), extended-precision add, subtract, extended-precision subtract, minimum (IMNMX), maximum instructions.

  - I32 (Mul)

    Estimated workload for all 32-bit integer multiply (IMUL), multiply-add (IMAD), extended-precision multiply-add, sum of absolute difference (ISAD), population count (POPC), count of leading zeros, most significant non-sign bit (FLO).

  - I32 (Shift)

    Estimated workload for all 32-bit integer shift left (SHL), shift right (SHR), funnel shift (SHF) instructions.

  - Cmp/Min/Max

    Estimated workload for all comparison operations

  - I32 (Bitfield/Rev)

    Estimated workload for all 32-bit integer bit reverse, bit field extract (BFE), and bit field insert (BFI) instructions

  - I32 (Bitwise Logic)

    Estimated workload for all logical operations (LOP).

  - Warp Shuffle

    Estimated workload for all warp shuffle (SHFL) instructions.

  - Video SIMD

    Estimated workload for all video vector instructions

  - Conv (From I8/I16 to I32)

    Estimated workload for all type conversions from 8-bit and 16-bit integer to 32-bit types (subset of I2I).

– Conv (To/From FP64)

Estimated workload for all type conversions from and to 64-bit types (subset of I2F, F2I, and F2F).

– Conv (All Other)

Estimated workload for all all other type conversions (remaining subset of I2I, I2F, F2I, and F2F).

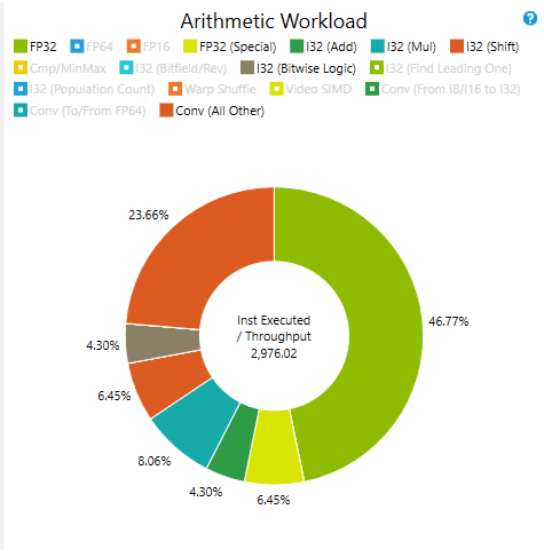

Figure 4.16: Arithmetic Workload chart example

4.1.9 Memory statistics

- Overview Chart

  Shows a summary view of the memory hierarchy of the CUDA programming model. Key metrics are reported for the areas that were covered by memory experiments during the data collection. The nodes in the diagram depict either a logical memory space (global, local, shared, ...) or an actual hardware unit on the chip (caches, shared memory, device memory). For the various caches the reported percentage number states the cache hit rate; that is the ratio of requests that could be served with data locally available to the cache over all requests made. Requests that hit data in the cache are served much faster than requests that miss the cache; missed data needs to be fetched from another layer of the memory hierarchy.

  Links between the nodes in the diagram depict the data paths between the SMs to the memory spaces into the memory system.

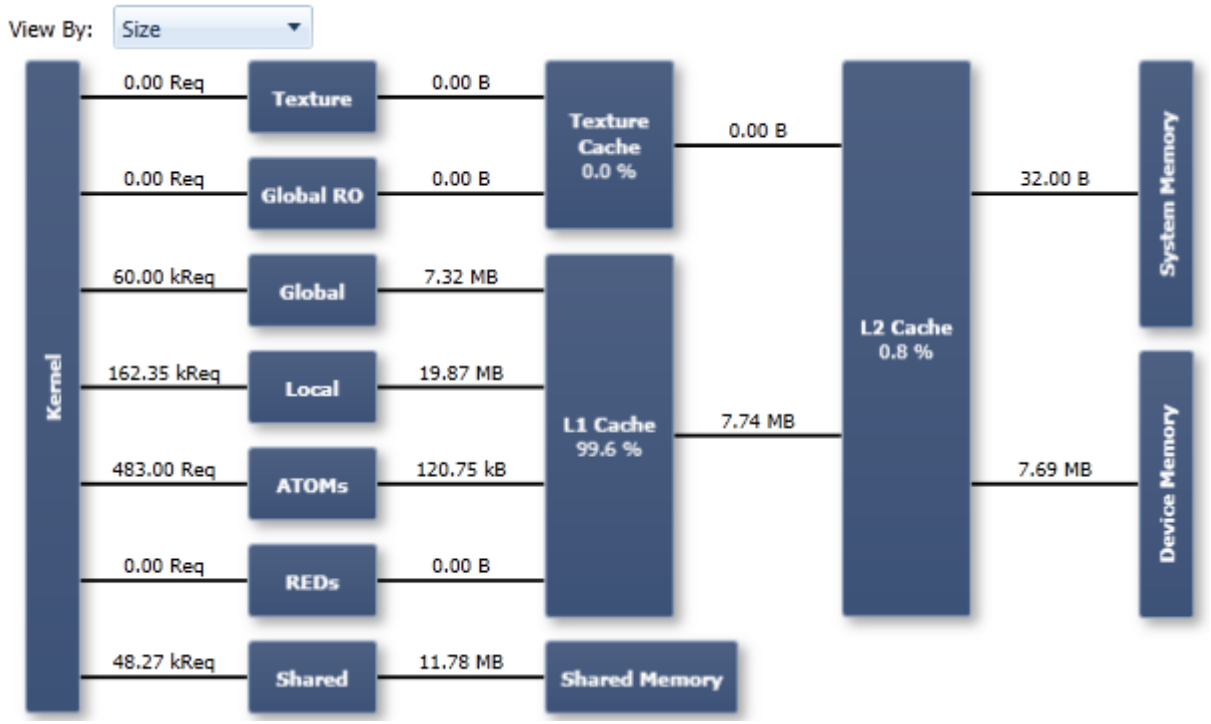

Figure 4.17: Memory Statistics chart example

- Shared memory Chart

  The number of Load/Store Requests equals the amount of shared memory instructions executed. When a warp executes an instruction that accesses shared memory, it resolves the bank conflicts. Each bank conflict forces a new memory transaction. The more transactions are necessary, the more unused words are transferred in addition to the words accessed by the threads, reducing the instruction throughput accordingly. Each memory transaction also requires the assembly instruction to be issued again; causing instruction replays if more than one transaction is required to fulfill the request of a warp.

  The Transactions Per Request chart shows the average number of shared memory transactions required per executed shared memory instruction, separately for load and store operations. Lower numbers are better; the target for a single shared memory operation on a full warp is 1 transaction for both, a 4byte access and an 8byte access, and 2 transactions for a 16byte access."

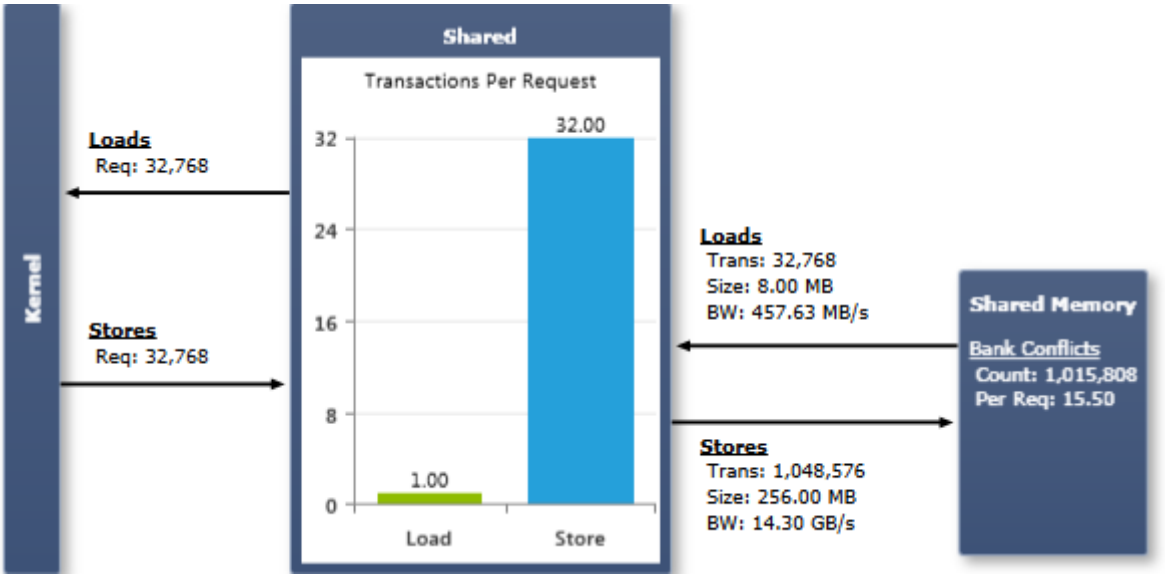

Figure 4.18: Shared Memory Statistics chart example

4.2 Equals

Figure 4.19 presents execution times of the function Equals in μs based on input's bitwise length. Below are charts generated by NSight.

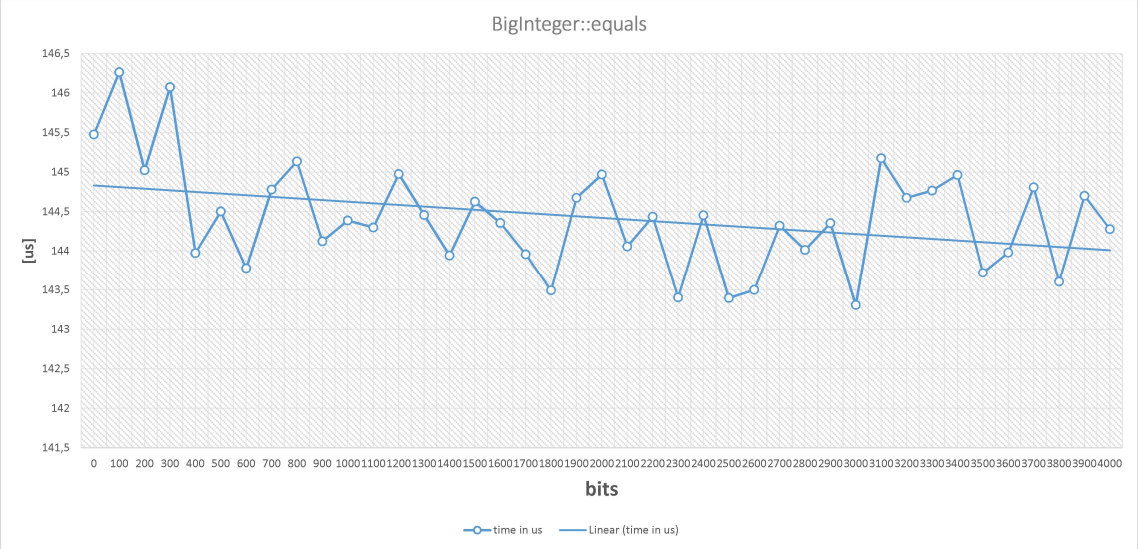

Figure 4.19: Execution time of function Equals in μs depending on input's bitwise length

Variable	Achieved	Theoretical	Device Limit
Occupancy Per SM			
Active Blocks		2	32
Active Warps	0.50	8	64
Active Threads		256	2048
Occupancy	0.78 %	12.50 %	100.00 %
Warps			
Threads/Block		128	1024
Warps/Block		4	32
Block Limit		16	32
Registers			
Registers/Thread		11	255
Registers/Block		2048	65536
Registers/SM		4096	65536
Block Limit		32	32
Shared Memory			
Shared Memory/Block		34448	49152
Shared Memory/SM		68896	98304
Block Limit		2	32

Figure 4.20: Occupancy statistics of Equals function

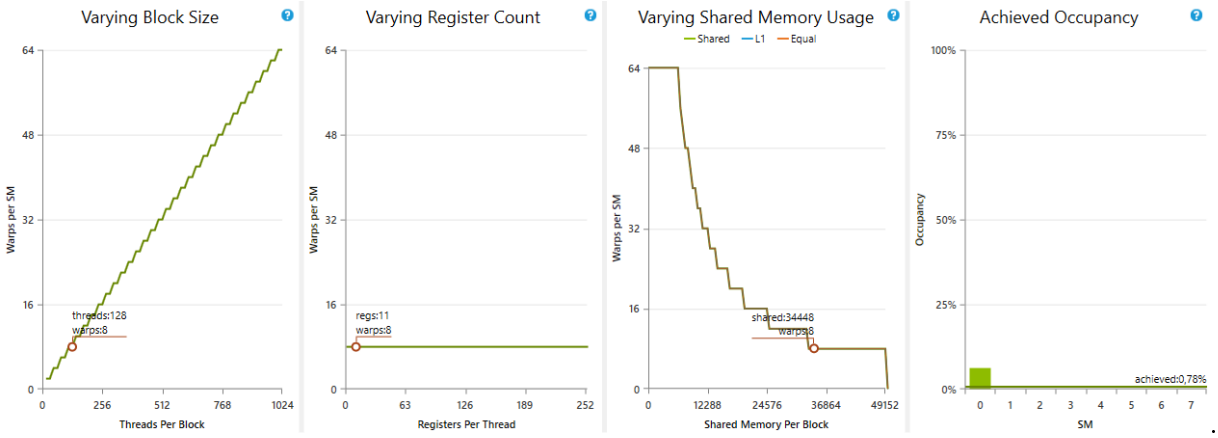

Figure 4.21: Occupancy charts of Equals function

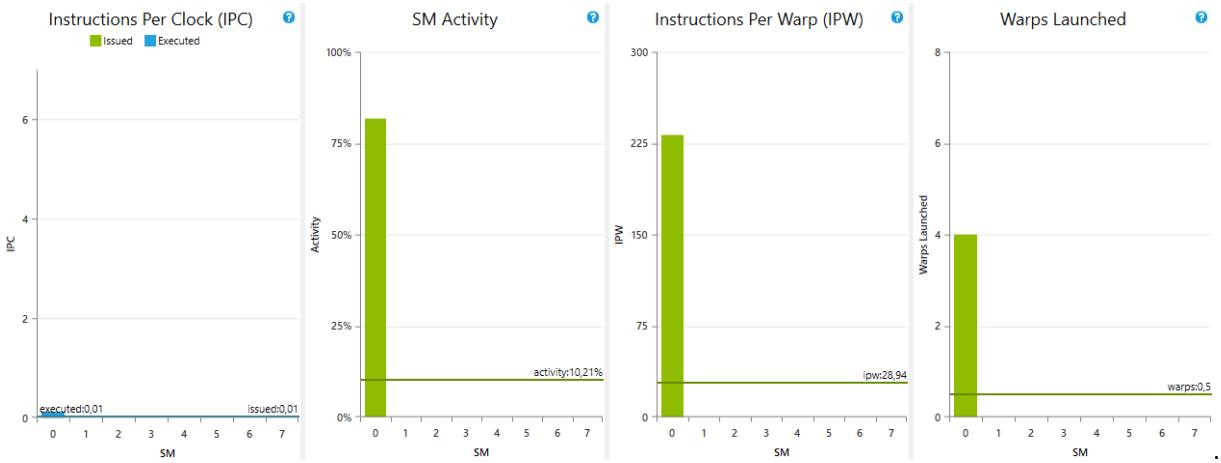

Figure 4.22: Instruction statistics of Equals function

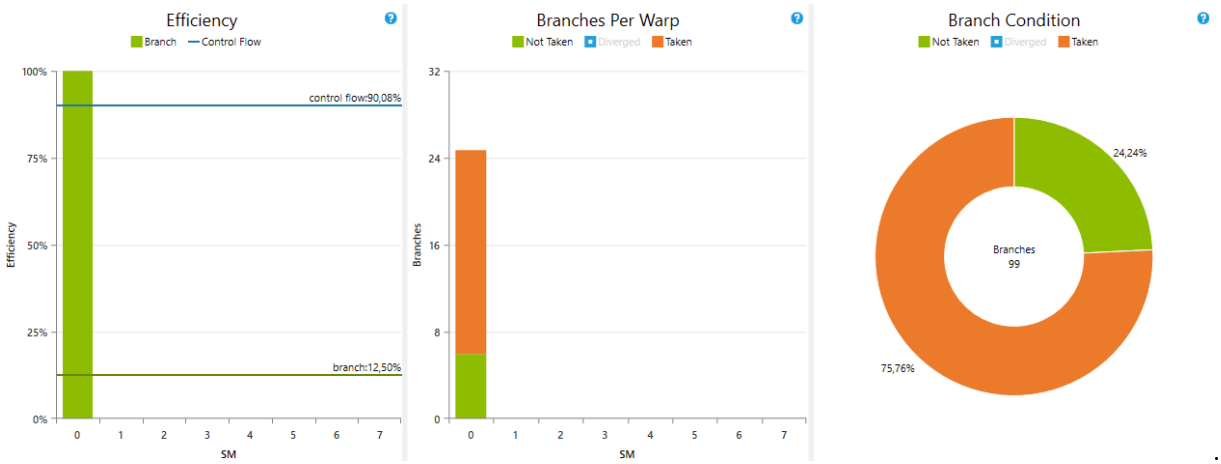

Figure 4.23: Branch statistics of Equals function

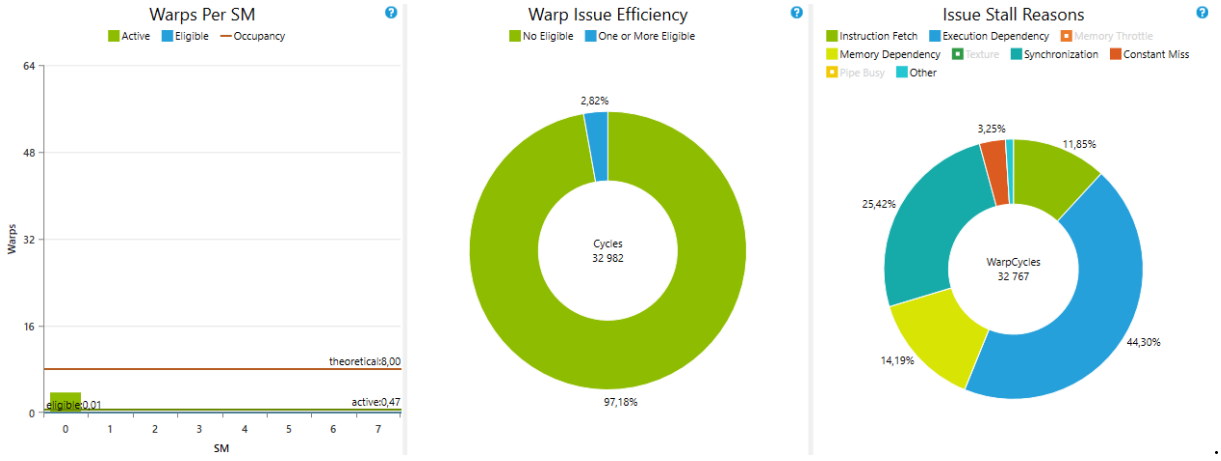

Figure 4.24: Issue efficiency of Equals function

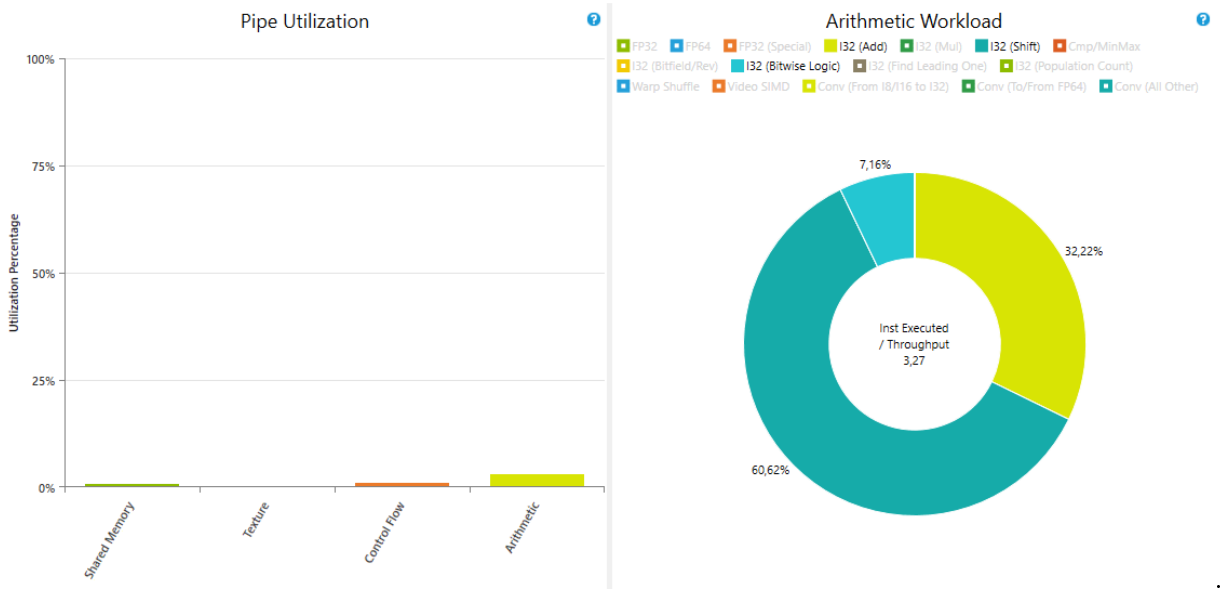

Figure 4.25: Pipe utilization of Equals function

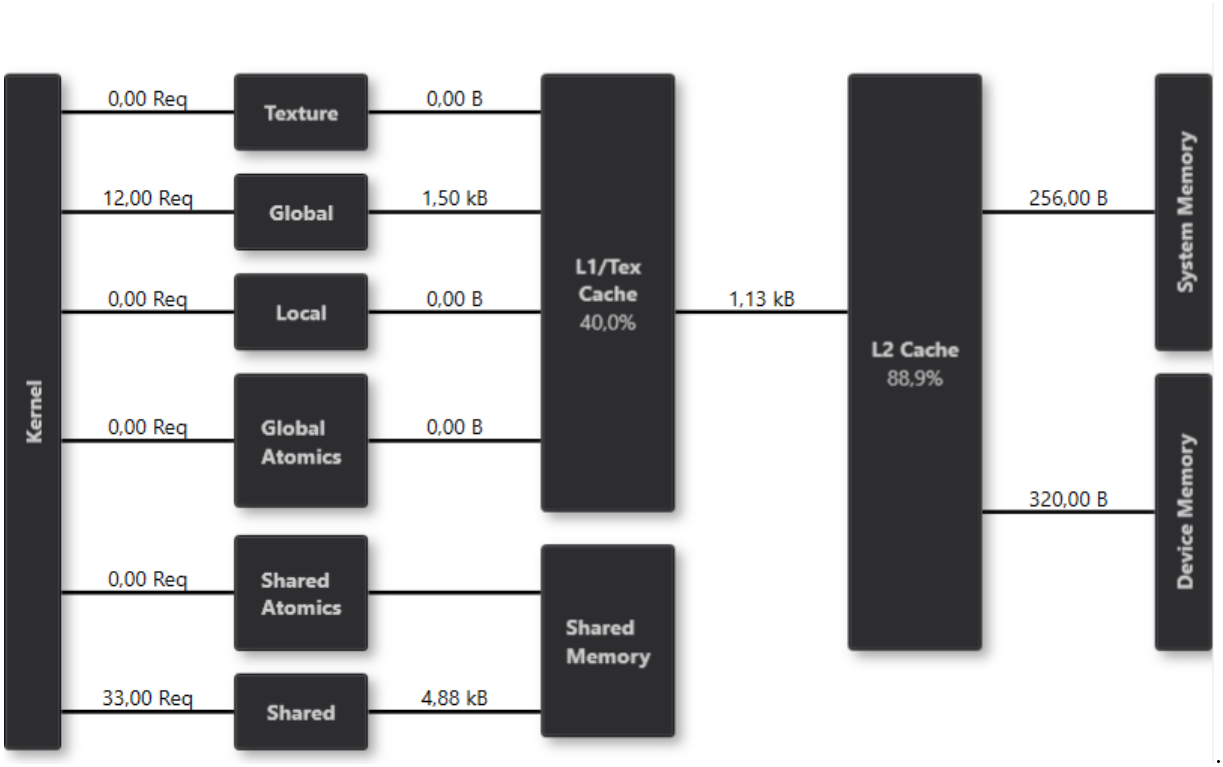

Figure 4.26: Memory overview of Equals function

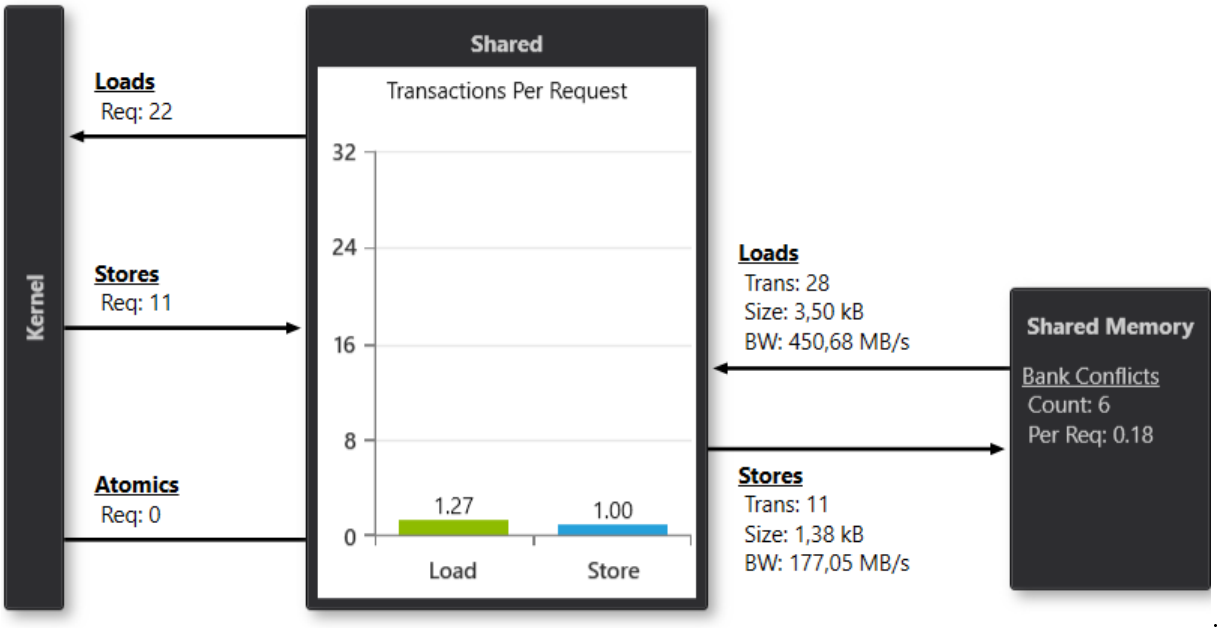

Figure 4.27: Shared memory chart of Equals function

Analyzing figure 4.19 we can see the trend line showing the average time actually drops as the input's length rises. This is probably caused by caching techniques implemented on the device. It is safe to state the function runs in constant time, and is resistant to timing attacks. Achieved occupancy at only 0.78% is caused by shared memory. In order to eliminate bank conflicts, the function uses a 7168 bytes of shared memory, where most of it is padding.

```
__shared__ int shared[7][256];
```

This successfully eliminates most of the bank conflicts,4.27 but reduces the active blocks count to two.4.21 Varying Block Size chart shows the occupancy could easily get higher by increasing the block size, but the algorithm is designed to operate and synchronizes on 128 threads. Solution of bank conflicts and lack of any atomic or floating point instructions in the function's code yields very good IPC coverage. Every issued instruction is immediately executed as shown on the Instruction Per Clock chart.4.22 The Branch Statistics4.23 are very good as well. No diverged threads gives 100% efficiency and supports resistance to side-channel. Despite this, the statistics on the Issue Efficiency charts look much worse.4.24 Very little eligible warps mostly stalled by execution dependency or synchronization indicate low performance of the code.

4.3 Compare

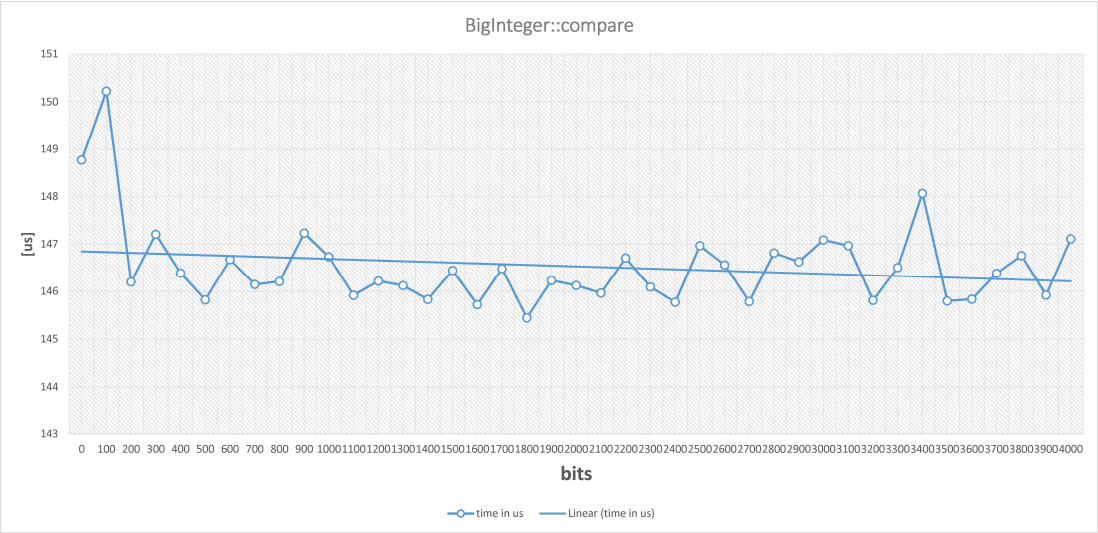

Figure 4.28: Execution time of function Compare in μs depending on input's bitwise length

Compare function contains very similar code to Equals. The main difference is - here the corresponding x and y values are not "xored" together, but rather compared to eventually distinguish greater value. The statistics only differ in Active Block count, which resulted in slightly higher occupancy - 18.75% theoretical.

	Variable	Achieved	Theoretical	Device Limit	
▲	Occupancy Per SM				
	Active Blocks		3	32	0 10 20 30
	Active Warps	0.50	12	64	0 20 40 60
	Active Threads		384	2048	0 1000 2000
	Occupancy	0.78 %	18.75 %	100.00 %	0% 50% 100%
▲	Warps				
	Threads/Block		128	1024	0 500 1000
	Warps/Block		4	32	0 10 20 30
	Block Limit		16	32	0 10 20 30
▲	Registers				
	Registers/Thread		23	255	0 100 200
	Registers/Block		3072	65536	0 20000 40000 60000
	Registers/SM		9216	65536	0 20000 40000 60000
	Block Limit		21	32	0 10 20 30
▲	Shared Memory				
	Shared Memory/Block		27280	49152	0 20000 40000
	Shared Memory/SM		81840	98304	0 50000
	Block Limit		3	32	0 10 20 30

Figure 4.29: Occupancy statistics of Compare function

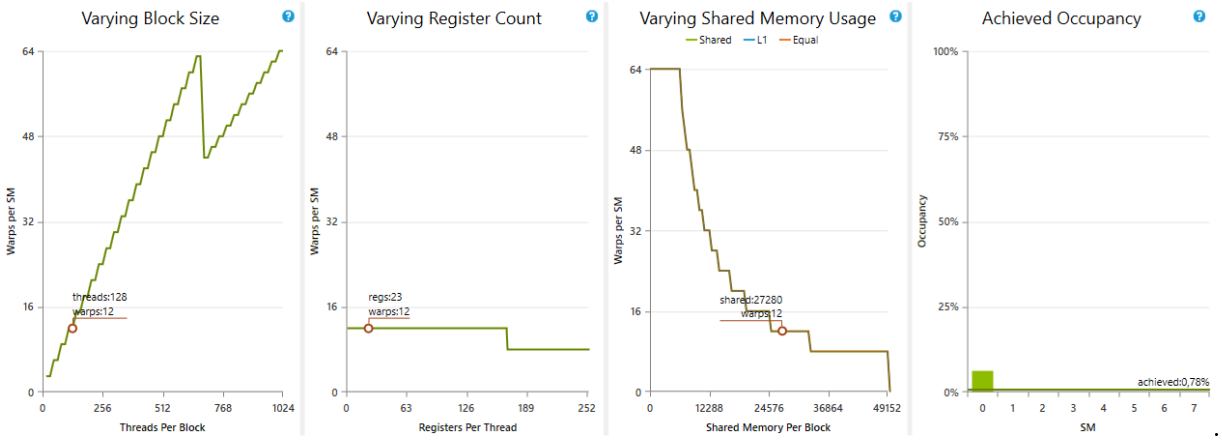

Figure 4.30: Occupancy charts of Compare function

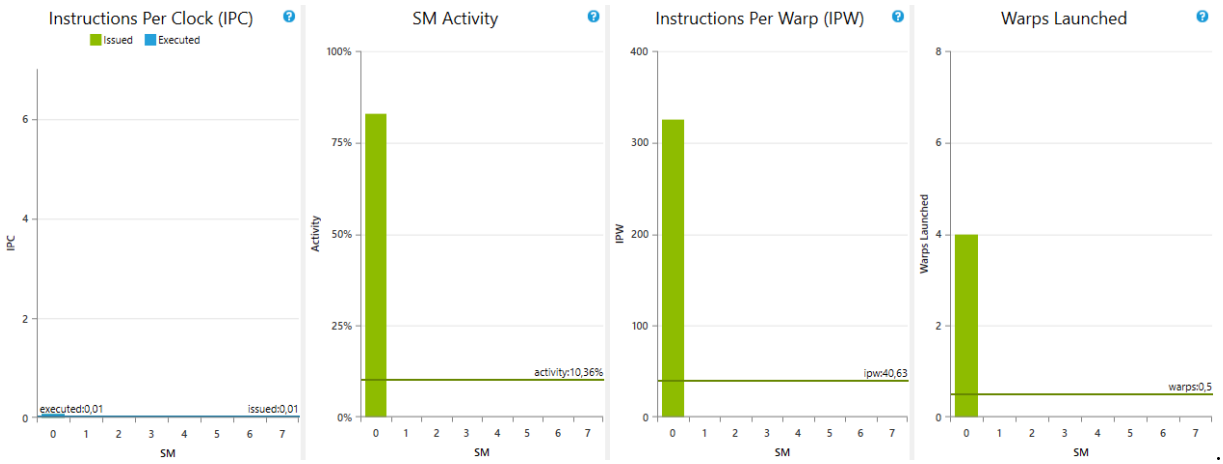

Figure 4.31: Instruction statistics of Compare function

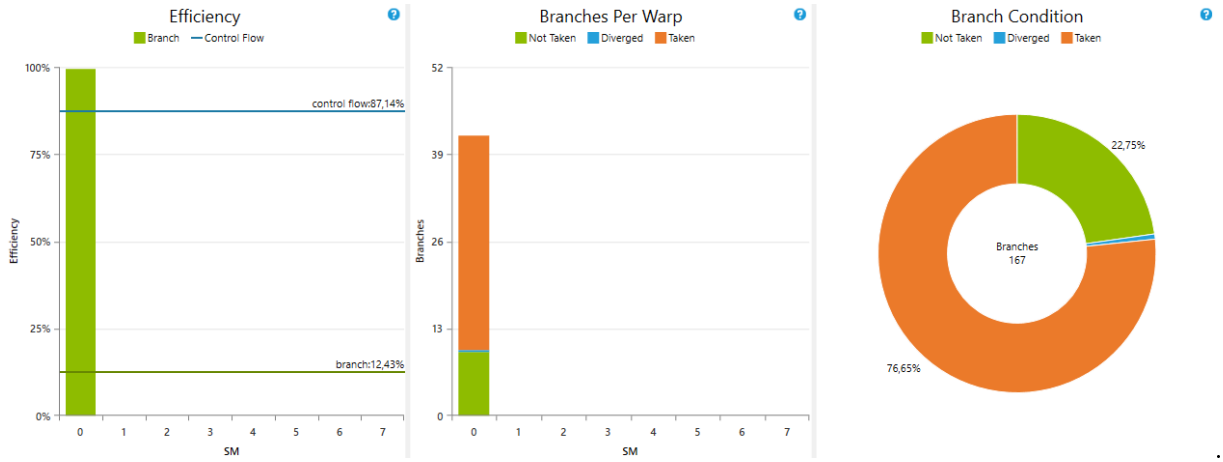

Figure 4.32: Branch statistics of Compare function

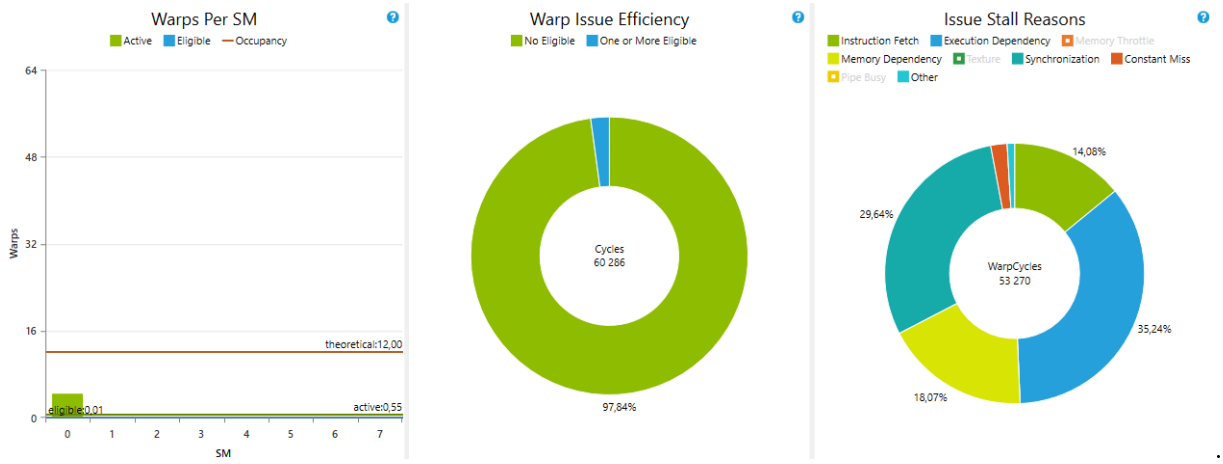

Figure 4.33: Issue efficiency of Compare function

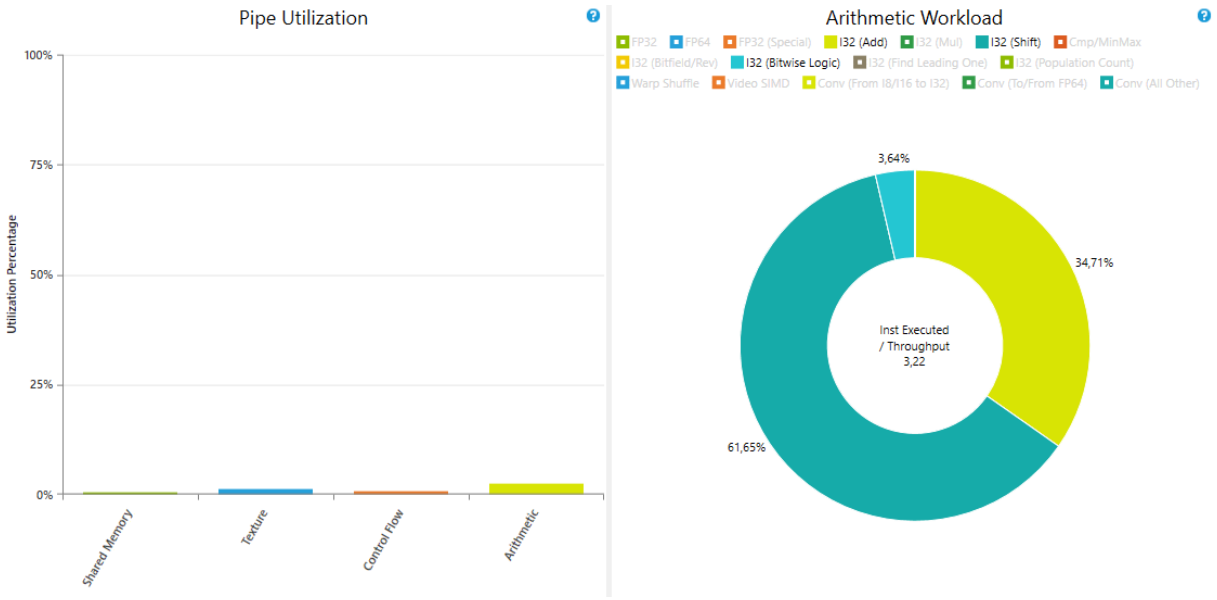

Figure 4.34: Pipe utilization of Compare function

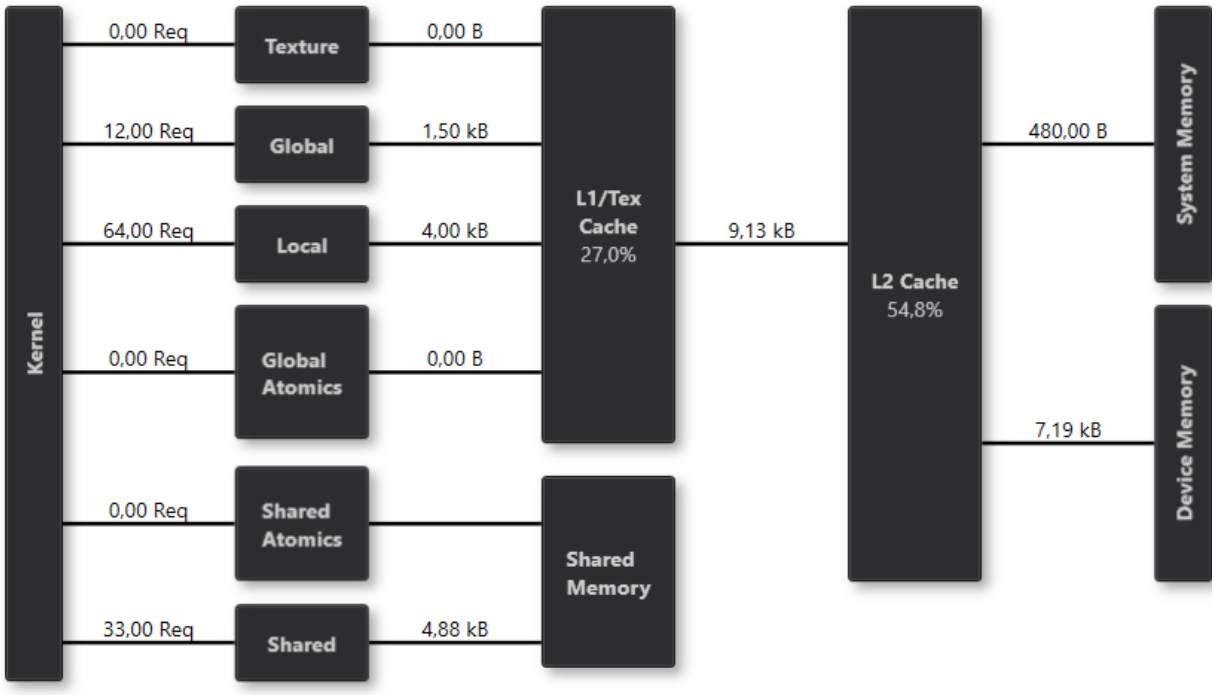

Figure 4.35: Memory overview of Compare function

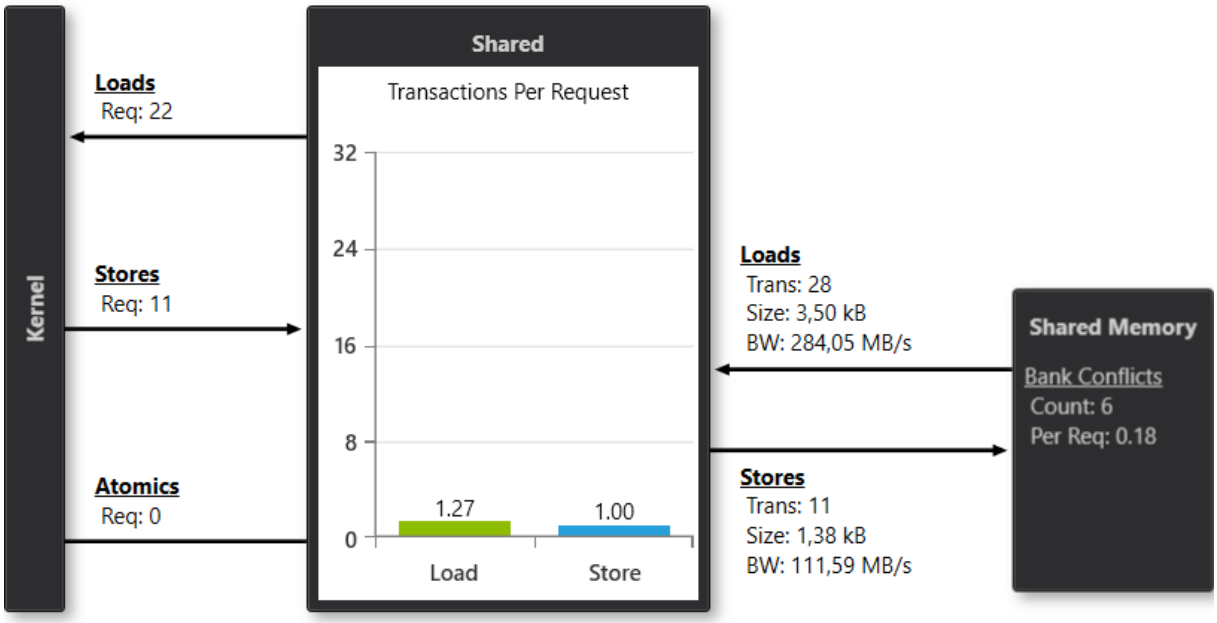

Figure 4.36: Shared memory chart of Compare function

4.4 Bit length

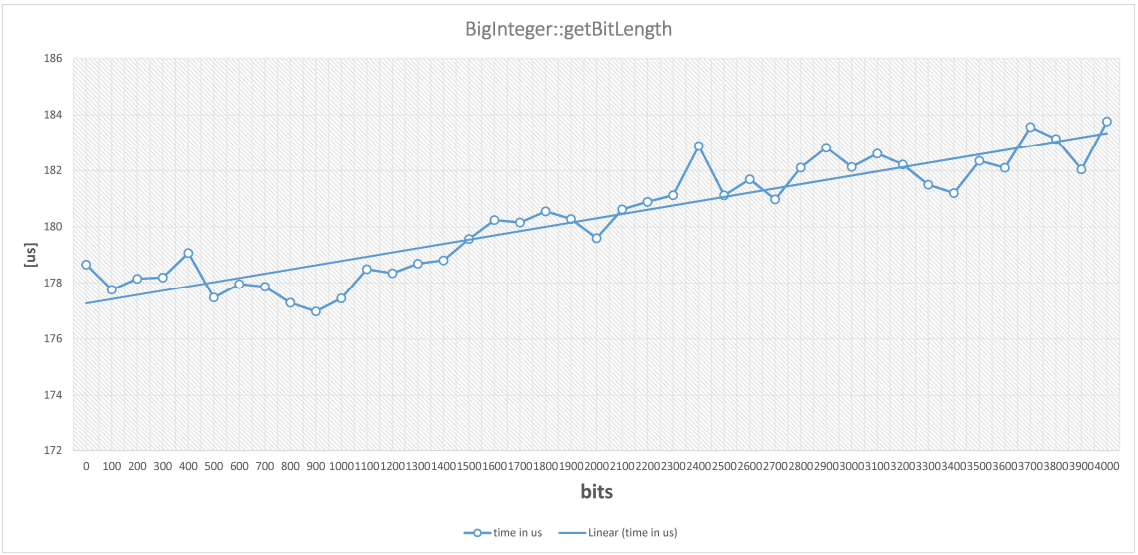

Figure 4.37: Execution time of function getBitLength in μs, depending on input's bitwise length

The function returning bitwise length of the array seems to be slightly dependent on the array's length. Even though the algorithm is fully designed to run in constant time, probably due to some compilers optimization, the execution time on the bigger input is longer by a little. The statistics look pretty much the same as in Compare function. A

bit less no eligible warps from which most are stalled by execution dependency.4.42 No bank conflicts whatsoever.4.45

Variable	Achieved	Theoretical	Device Limit	
Occupancy Per SM				
Active Blocks		3	32	0 10 20 30
Active Warps	0.50	12	64	0 20 40 60
Active Threads		384	2048	0 1000 2000
Occupancy	0.78 %	18.75 %	100.00 %	0% 50% 100%
Warps				
Threads/Block		128	1024	0 500 1000
Warps/Block		4	32	0 10 20 30
Block Limit		16	32	0 10 20 30
Registers				
Registers/Thread		12	255	0 100 200
Registers/Block		2048	65536	0 20000 40000 60000
Registers/SM		6144	65536	0 20000 40000 60000
Block Limit		32	32	0 10 20 30
Shared Memory				
Shared Memory/Block		27280	49152	0 20000 40000
Shared Memory/SM		81840	98304	0 50000
Block Limit		3	32	0 10 20 30

Figure 4.38: Occupancy statistics of getBitLength function

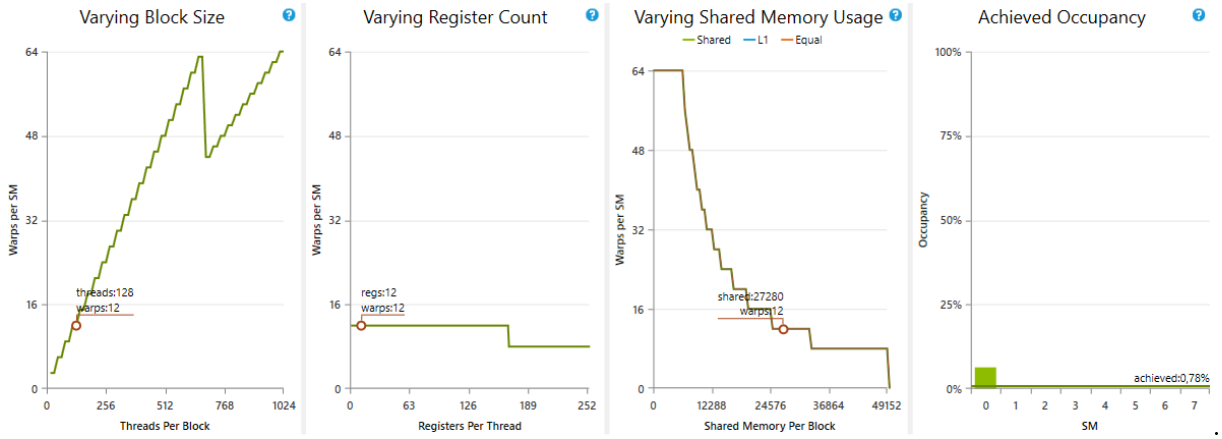

Figure 4.39: Occupancy charts of getBitLength function

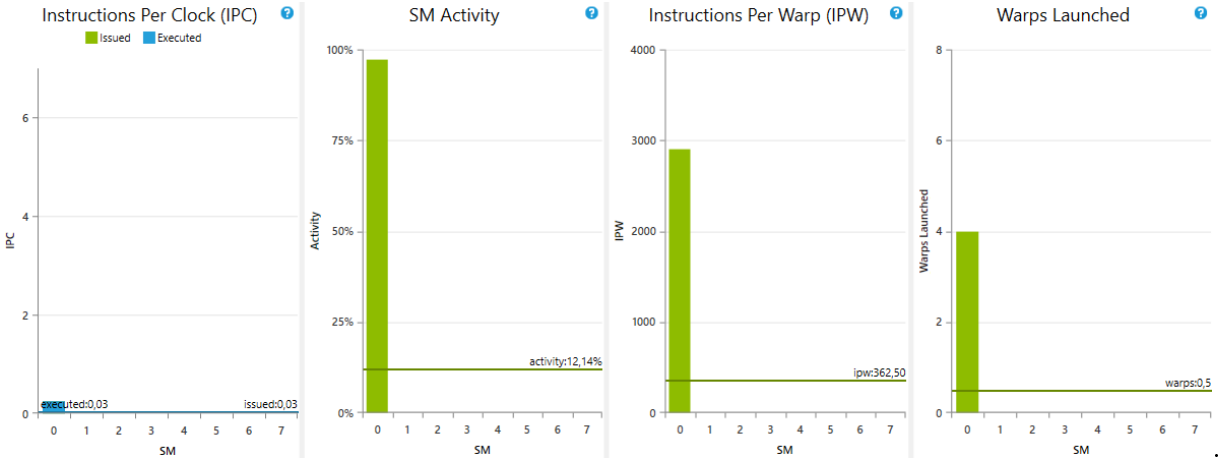

Figure 4.40: Instruction statistics of getBitLength function

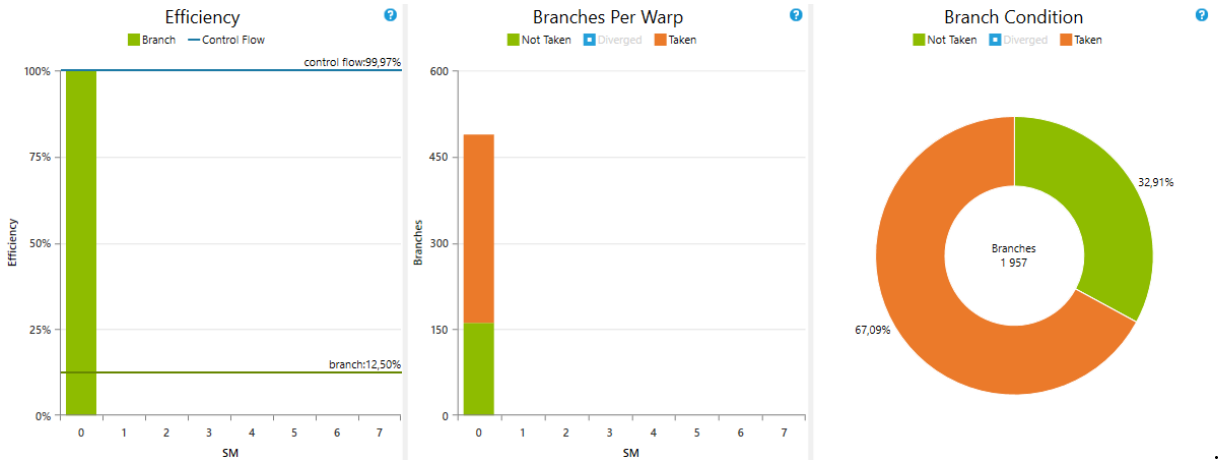

Figure 4.41: Branch statistics of getBitLength function

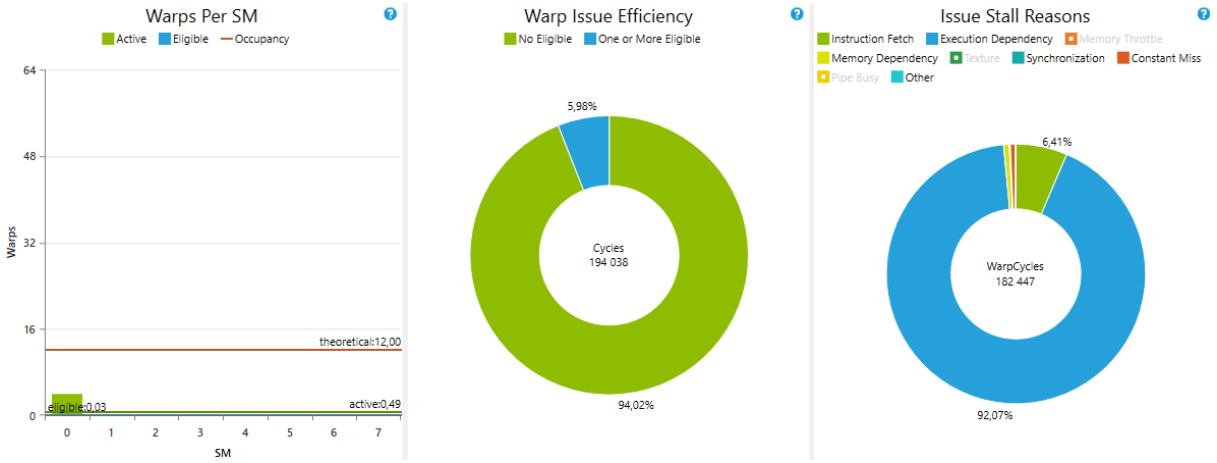

Figure 4.42: Issue efficiency of getBitLength function

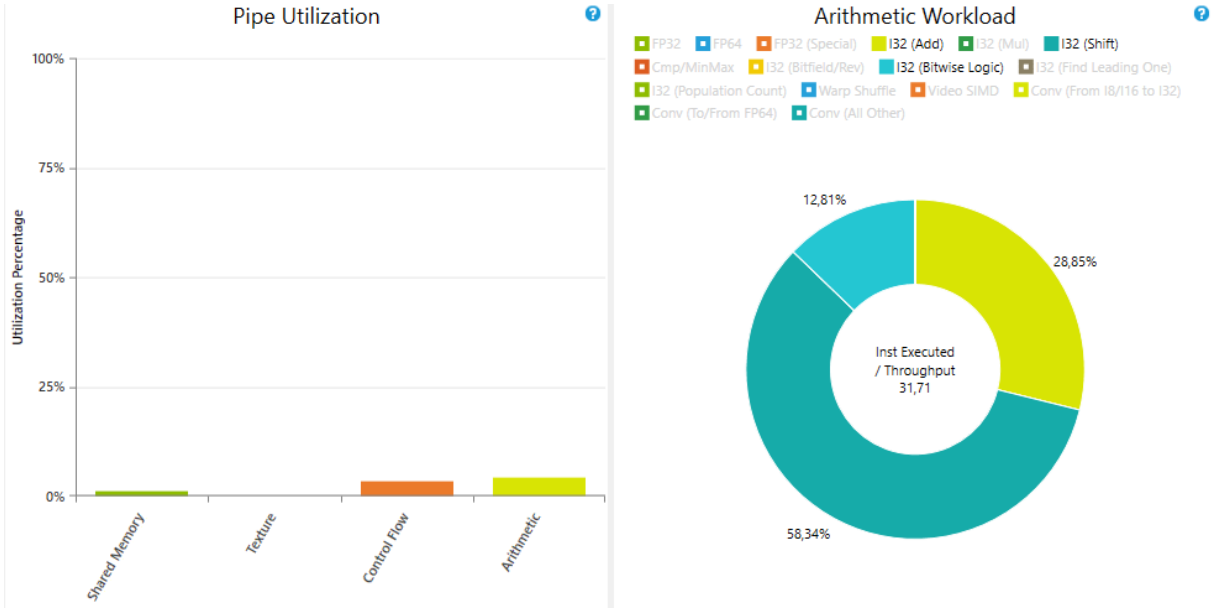

Figure 4.43: Pipe utilization of getBitLength function

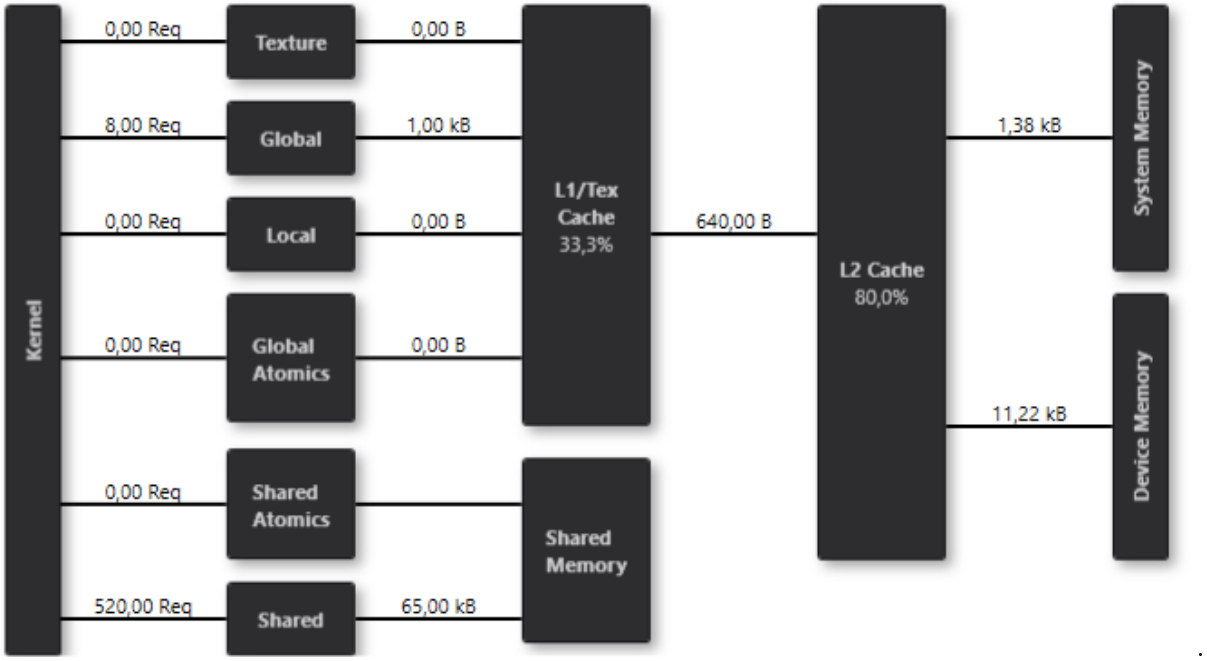

Figure 4.44: Memory overview of getBitLength function

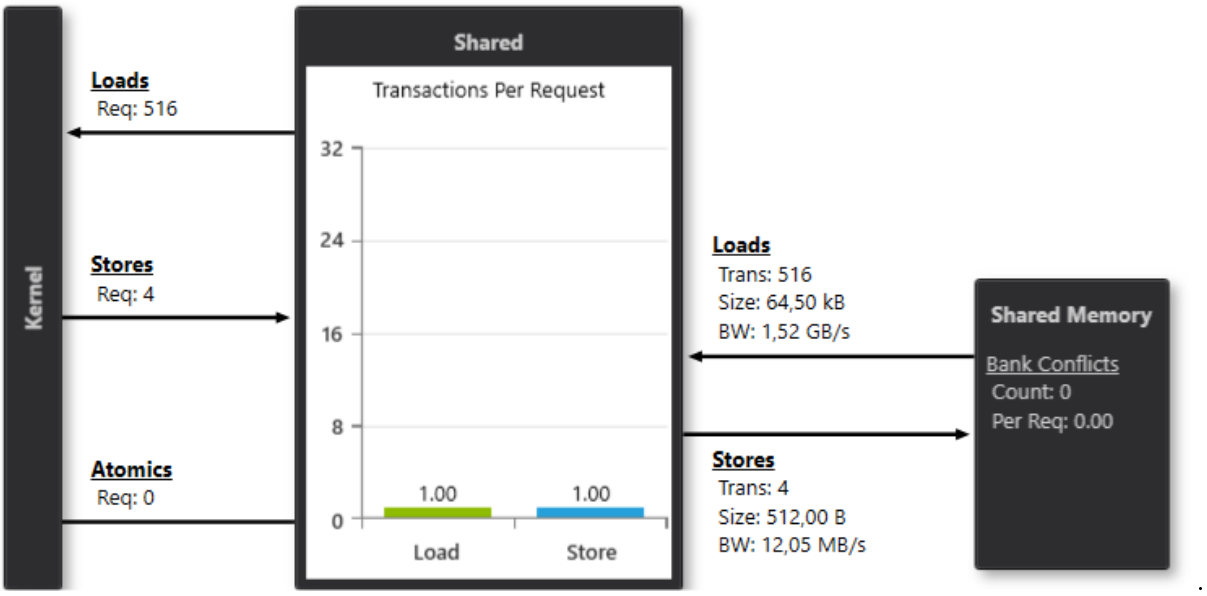

Figure 4.45: Shared memory chart of getBitLength function

4.5 Left shift

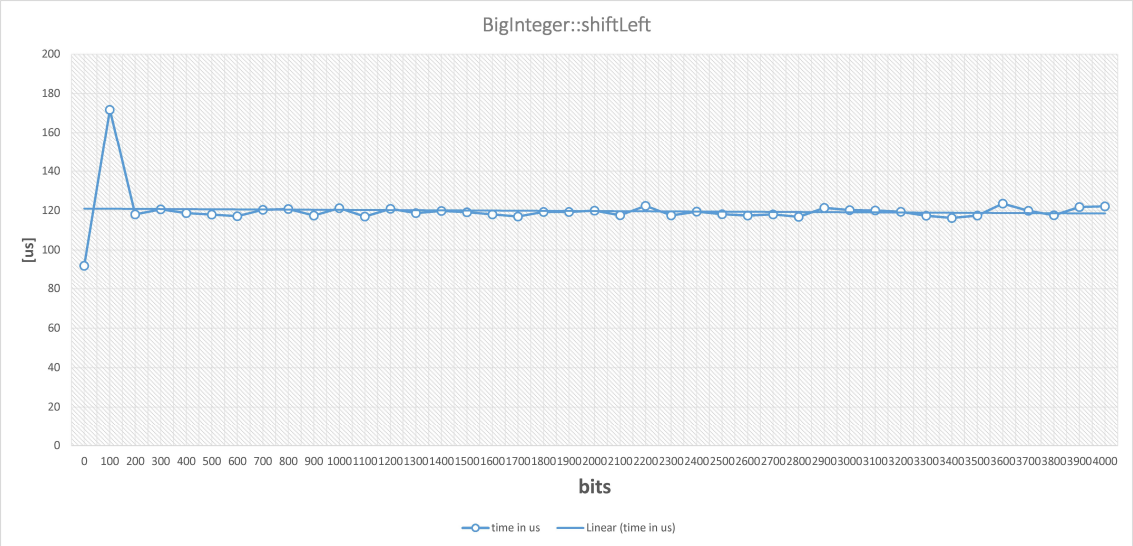

Figure 4.46: Execution time of function Left shift in μs, depending on number of bits to shift

The left shift function ideally operates in resistance to side-channel. The execution time is constant from small bit lengths up to 4000 bits, with only minute offsets. This is achieved by proper implementation for CUDA device execution, meaning there are very little branching conditions, no loops and most instructions are logics and simple integer arithmetics. The statistics are very similar to other functions designed for 128 threads (4 warps). Occupancy limited by memory to 18.75%, three active blocks, no bank conflicts, good branch statistics and poor warps issue efficiency.

Variable	Achieved	Theoretical	Device Limit	
Occupancy Per SM				
Active Blocks		3	32	0 10 20 30
Active Warps	0.50	12	64	0 20 40 60
Active Threads		384	2048	0 1000 2000
Occupancy	0.78 %	18.75 %	100.00 %	0% 50% 100%
Warps				
Threads/Block		128	1024	0 500 1000
Warps/Block		4	32	0 10 20 30
Block Limit		16	32	0 10 20 30
Registers				
Registers/Thread		20	255	0 100 200
Registers/Block		3072	65536	0 20000 40000 60000
Registers/SM		9216	65536	0 20000 40000 60000
Block Limit		21	32	0 10 20 30
Shared Memory				
Shared Memory/Block		27280	49152	0 20000 40000
Shared Memory/SM		81840	98304	0 50000
Block Limit		3	32	0 10 20 30

Figure 4.47: Occupancy statistics of Left shift function

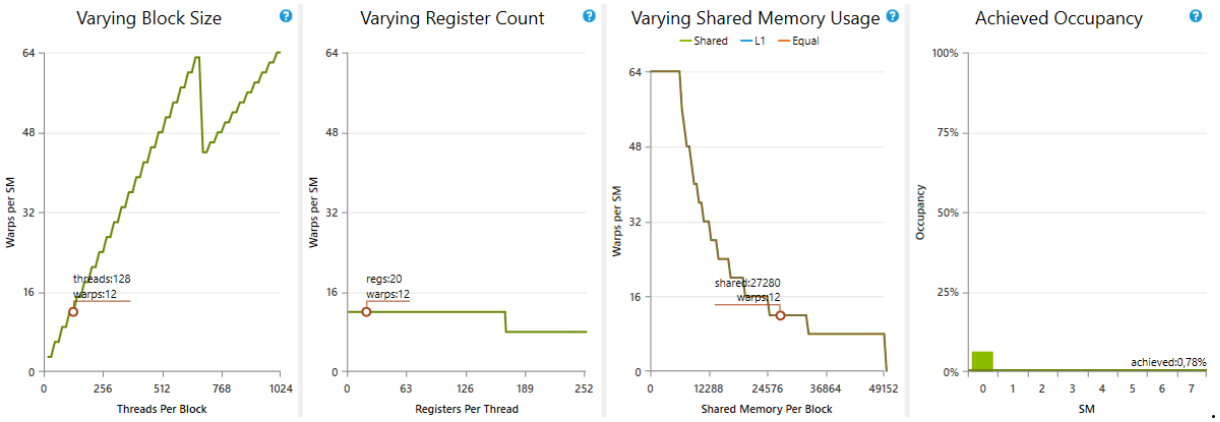

Figure 4.48: Occupancy charts of Left shift function

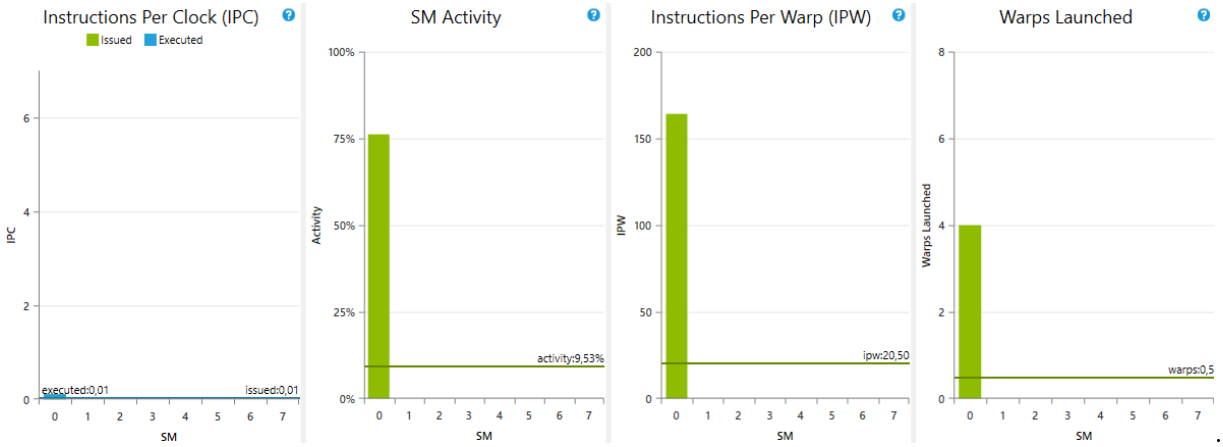

Figure 4.49: Instruction statistics of Left shift function

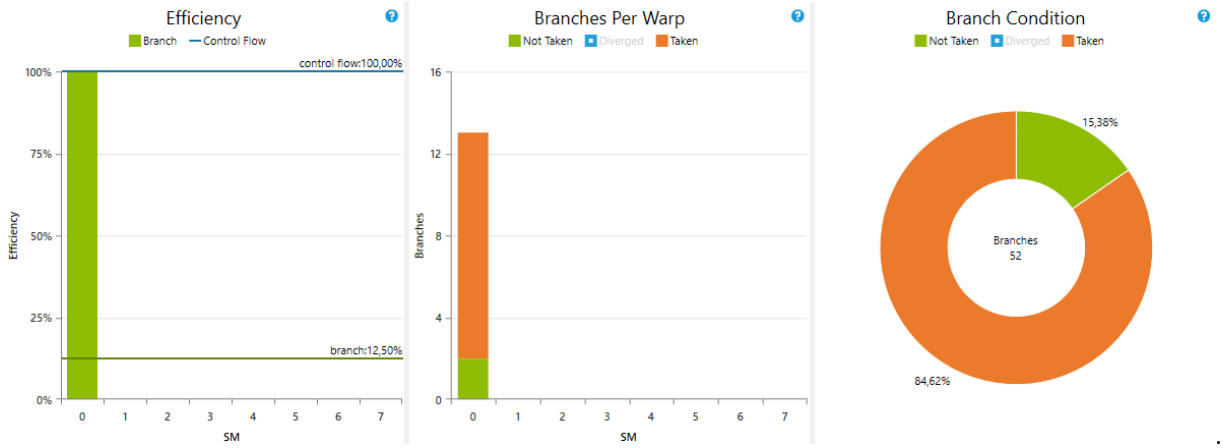

Figure 4.50: Branch statistics of Left shift function

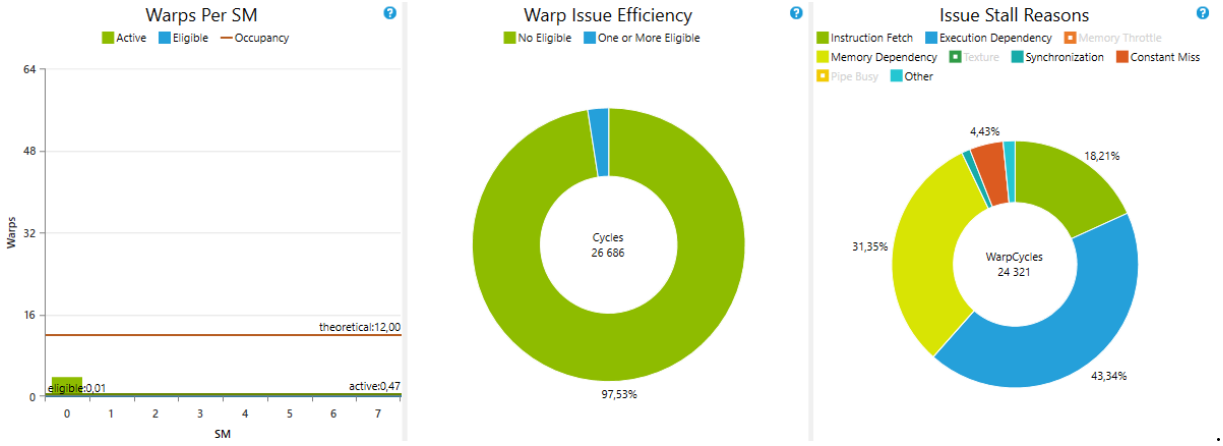

Figure 4.51: Issue efficiency of Left shift function

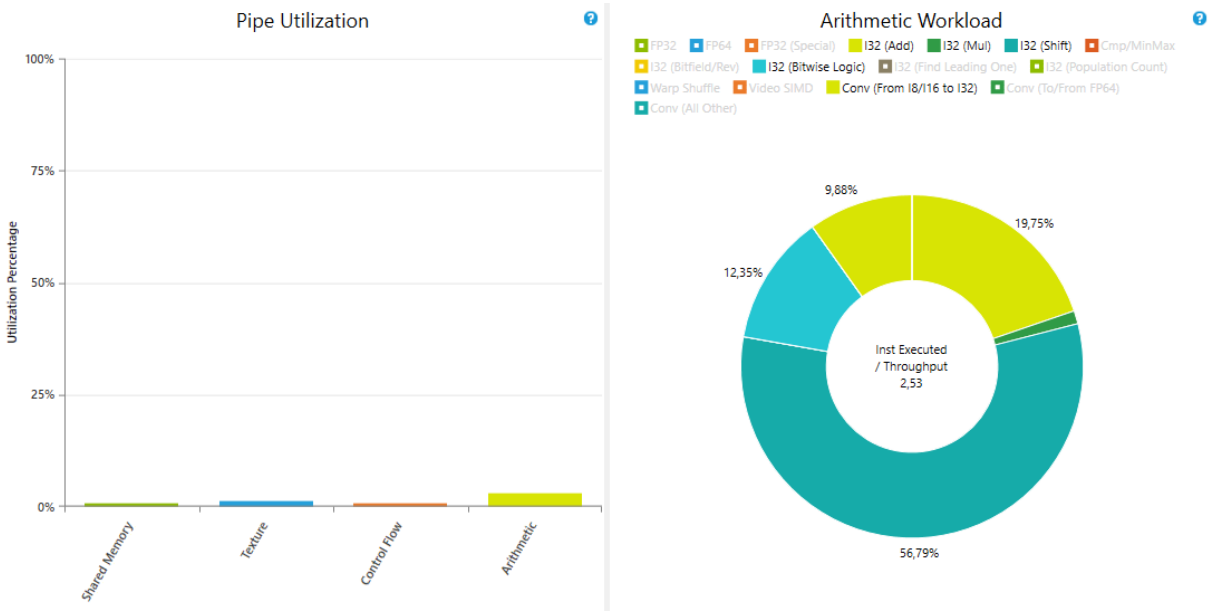

Figure 4.52: Pipe utilization of Left shift function

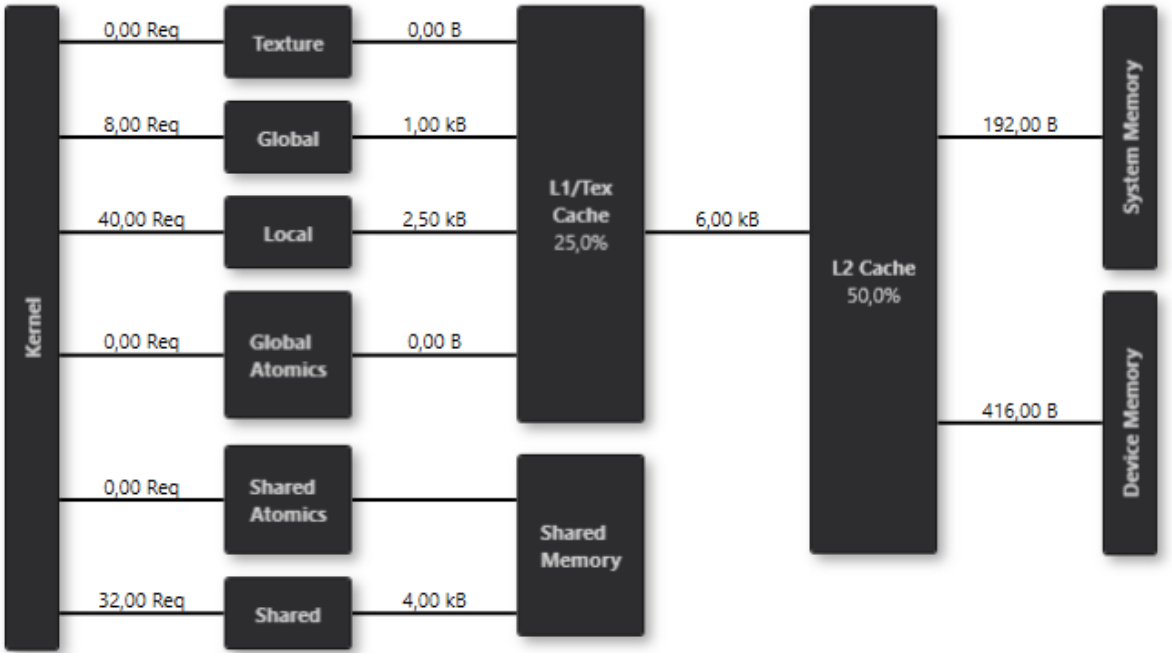

Figure 4.53: Memory overview of Left shift function

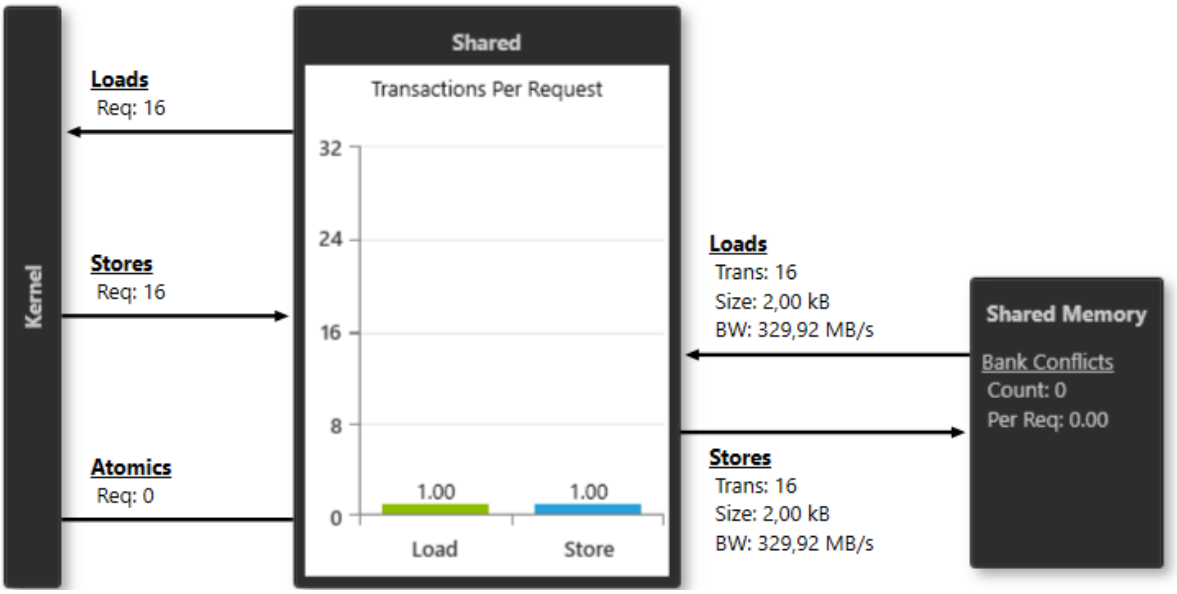

Figure 4.54: Shared memory chart of Left shift function

4.6 Right shift

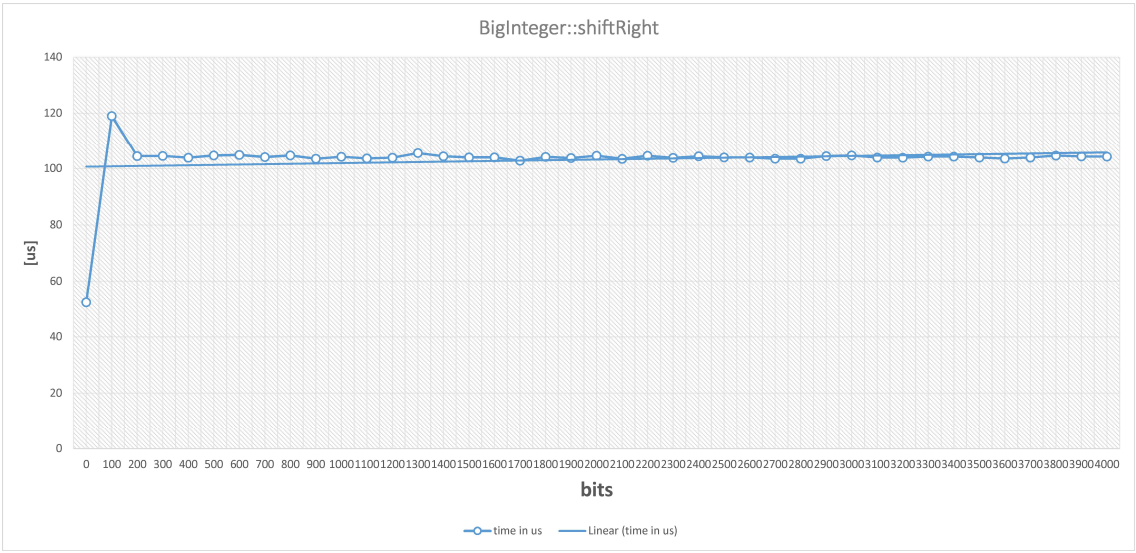

Figure 4.55: Execution time of function Right shift in μs, depending on number of bits to shift

The right shift function is very much the same as the left shift. Small difference in statics is shown in Branches per Warp chart, 4.59 most probably caused by code listed below.

```
if (index - 1 >= 0)
     sharedResult[index - 1] = sharedX[index] << remainingBits;
else
     sharedResult[127] = 0UL;
__syncthreads();
```

The condition blocks writing out of the array's bounds. The "else" statement occurs only for thread 0, which caused 3.33% diverged branches. This has a negligible effect on function's performance. The average execution time is roughly 20μs less then left shift function.

Variable	Achieved	Theoretical	Device Limit	
Occupancy Per SM				
Active Blocks		3	32	0 10 20 30
Active Warps	0.50	12	64	0 20 40 60
Active Threads		384	2048	0 1000 2000
Occupancy	0.78 %	18.75 %	100.00 %	0% 50% 100%
Warps				
Threads/Block		128	1024	0 500 1000
Warps/Block		4	32	0 10 20 30
Block Limit		16	32	0 10 20 30
Registers				
Registers/Thread		20	255	0 100 200
Registers/Block		3072	65536	0 20000 40000 60000
Registers/SM		9216	65536	0 20000 40000 60000
Block Limit		21	32	0 10 20 30
Shared Memory				
Shared Memory/Block		27280	49152	0 20000 40000
Shared Memory/SM		81840	98304	0 50000
Block Limit		3	32	0 10 20 30

Figure 4.56: Occupancy statistics of Right shift function

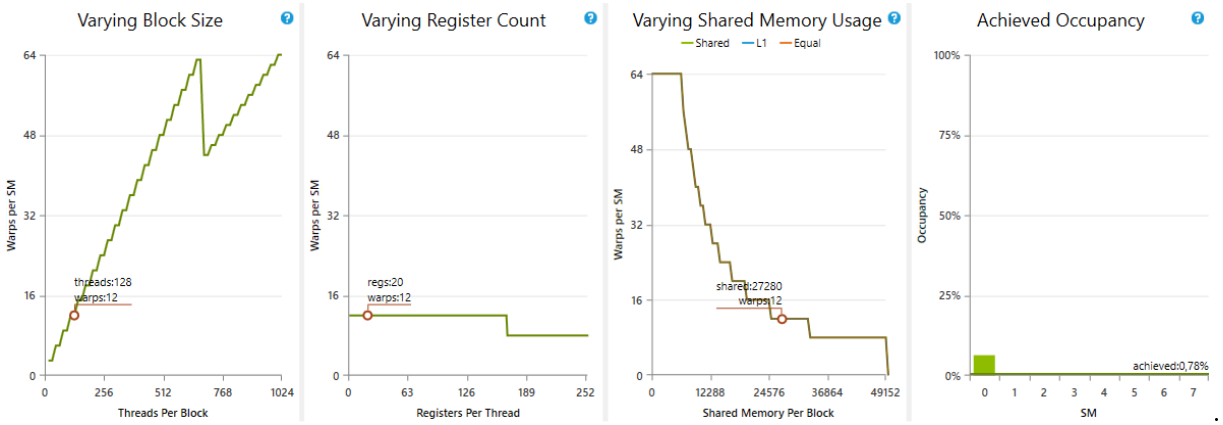

Figure 4.57: Occupancy charts of Right shift function

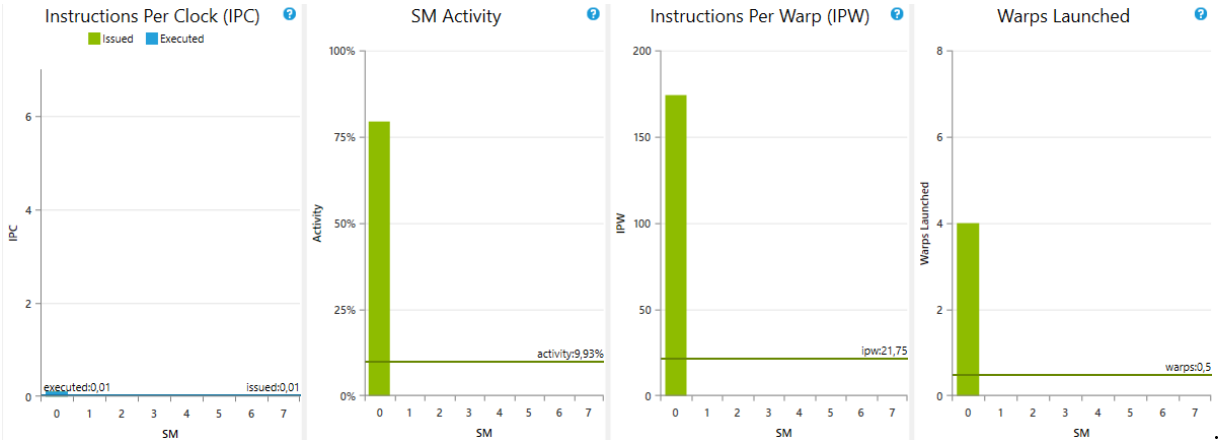

Figure 4.58: Instruction statistics of Right shift function

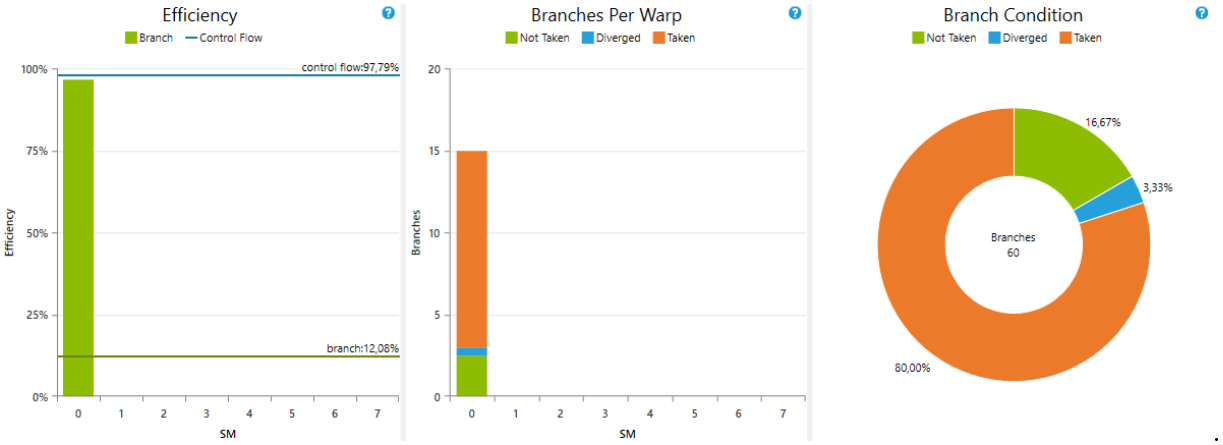

Figure 4.59: Branch statistics of Right shift function

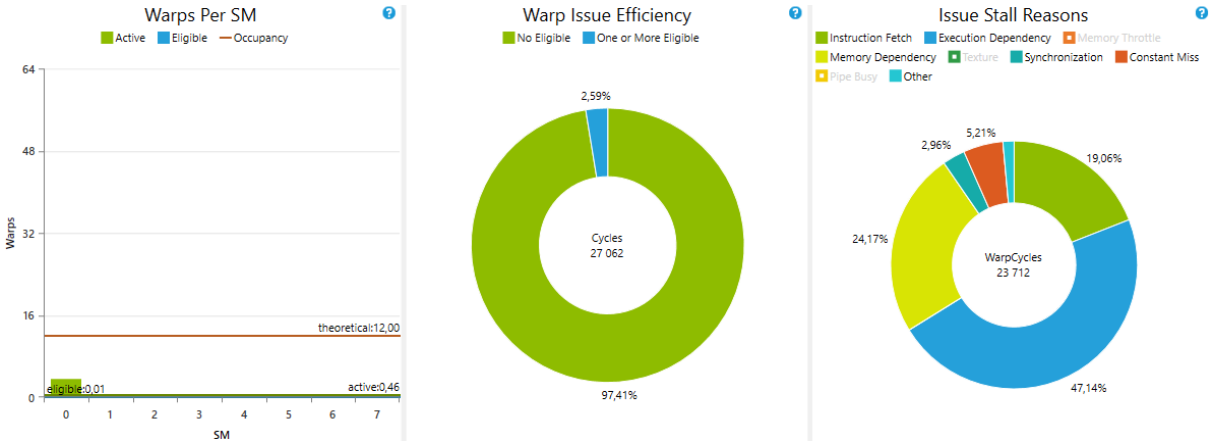

Figure 4.60: Issue efficiency of Right shift function

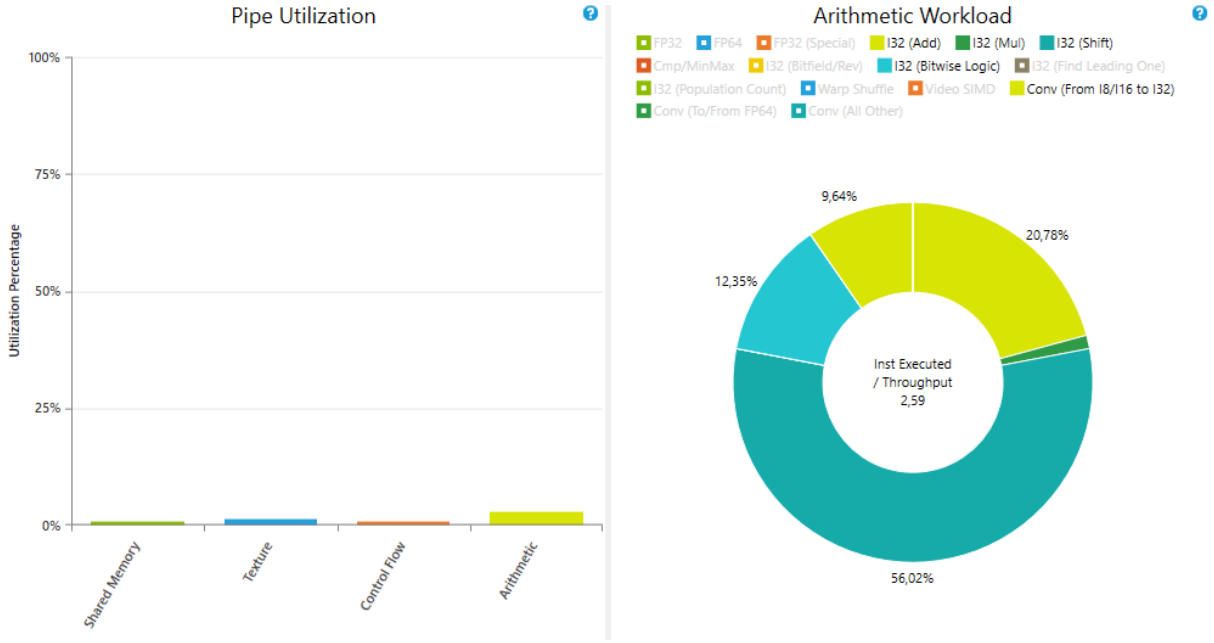

Figure 4.61: Pipe utilization of Right shift function

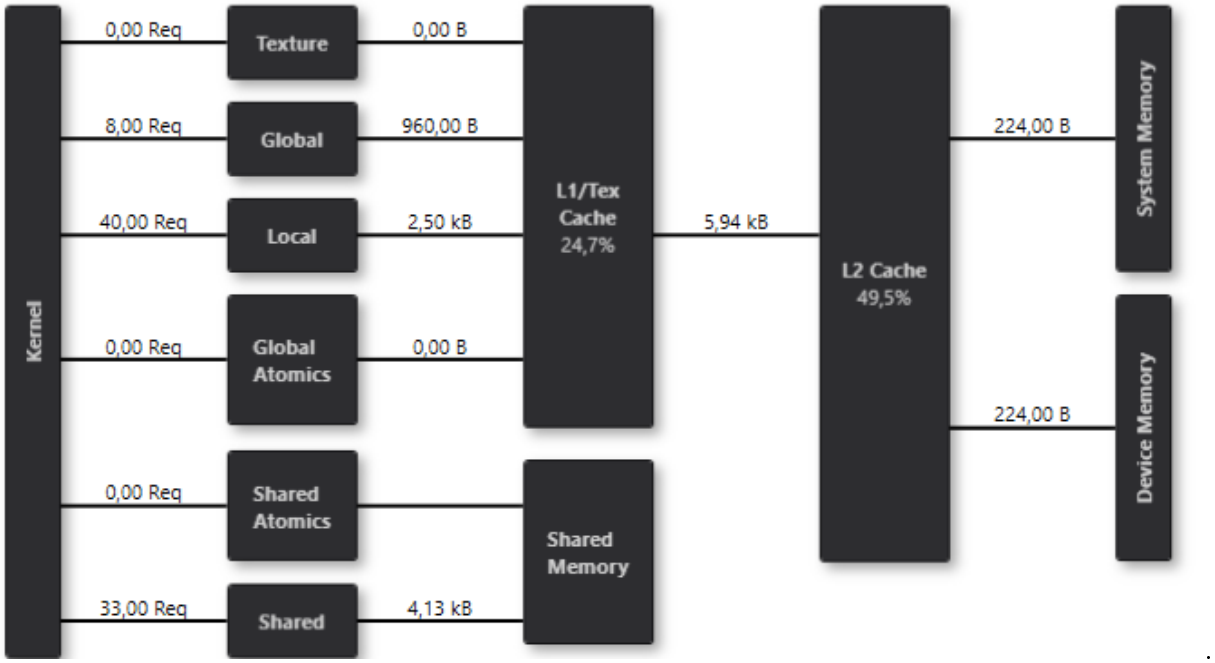

Figure 4.62: Memory overview of Right shift function

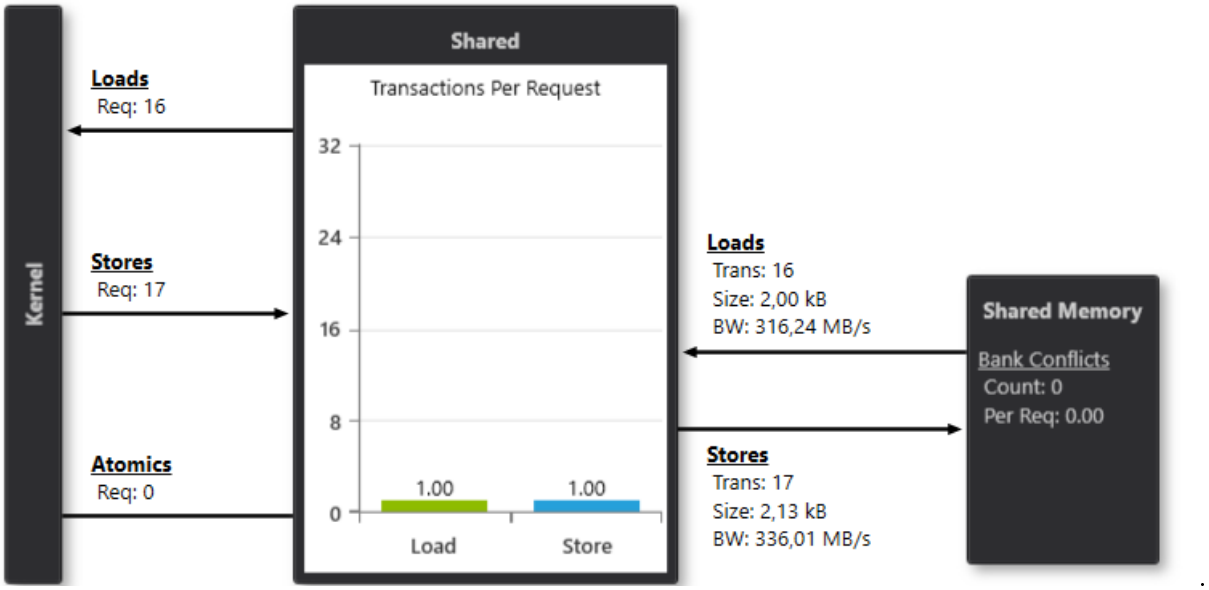

Figure 4.63: Shared memory chart of Right shift function

4.7 Add

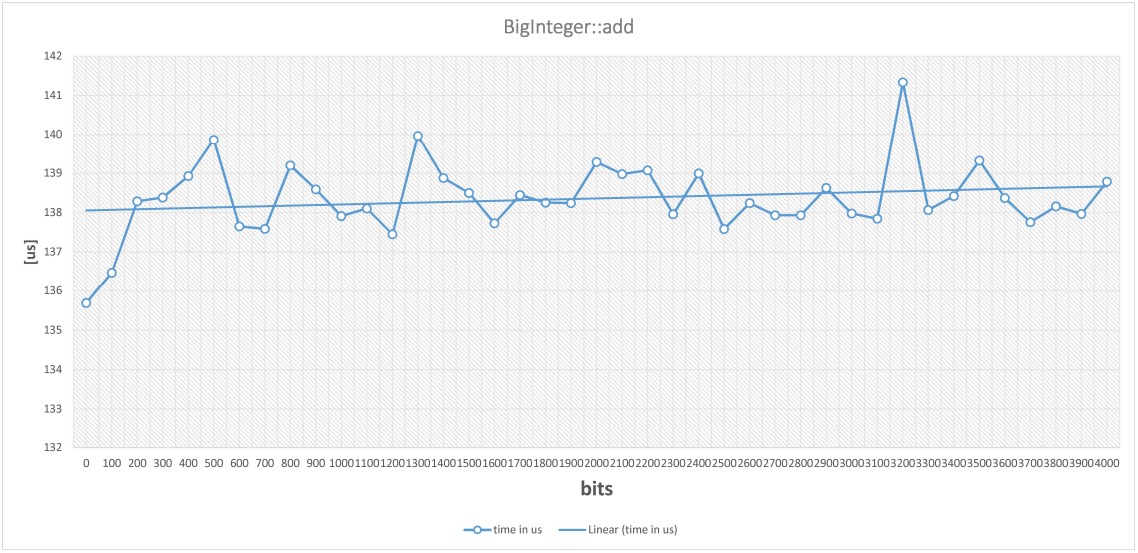

Figure 4.64: Execution time of function Add in μs depending on input's bitwise length

The Add function does not depend on the inputs' lengths as presented on the chart above, but the execution times are a bit scattered in comparison to left/right shifts. The function launches 1 warp (96 threads less then the functions described above), yet the statistics hardly changed.

Variable	Achieved	Theoretical	Device Limit
Occupancy Per SM			
Active Blocks		3	32
Active Warps	0.12	3	64
Active Threads		96	2048
Occupancy	0.20 %	4.69 %	100.00 %
Warps			
Threads/Block		32	1024
Warps/Block		1	32
Block Limit		64	32
Registers			
Registers/Thread		13	255
Registers/Block		512	65536
Registers/SM		1536	65536
Block Limit		128	32
Shared Memory			
Shared Memory/Block		27280	49152
Shared Memory/SM		81840	98304
Block Limit		3	32

Figure 4.65: Occupancy statistics of Add function

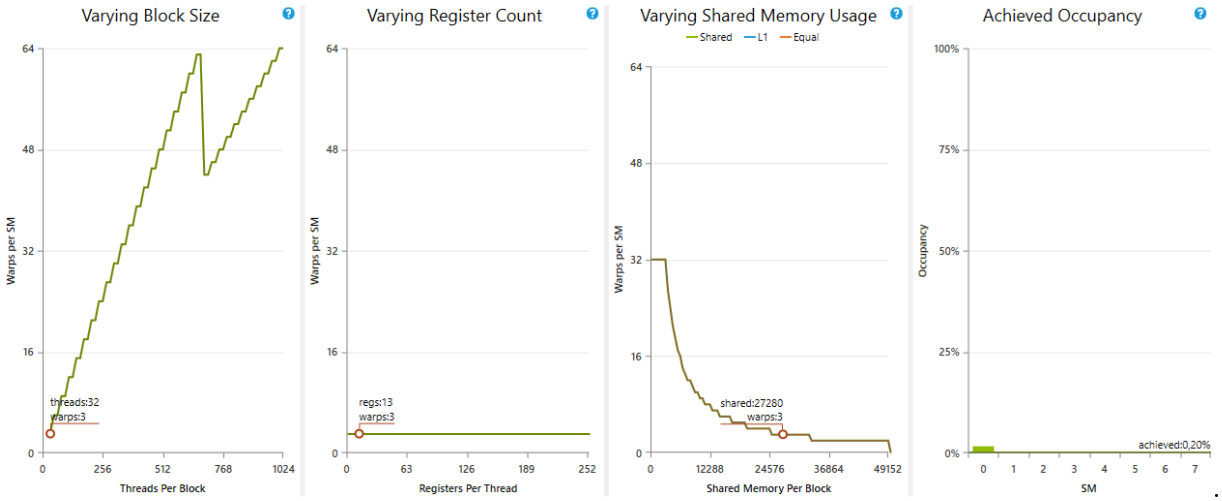

Figure 4.66: Occupancy charts of Add function

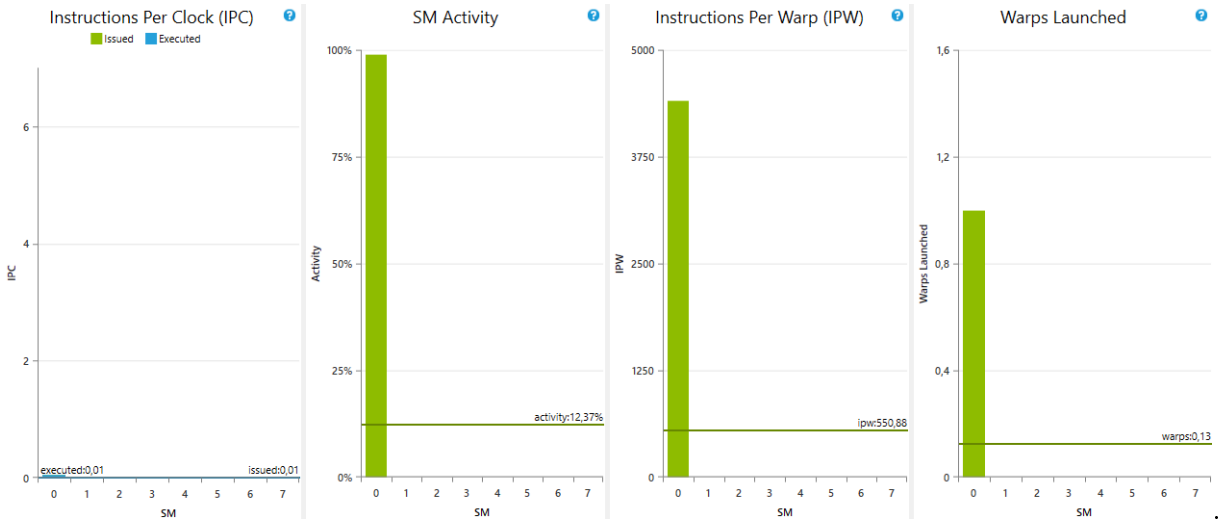

Figure 4.67: Instruction statistics of Add function

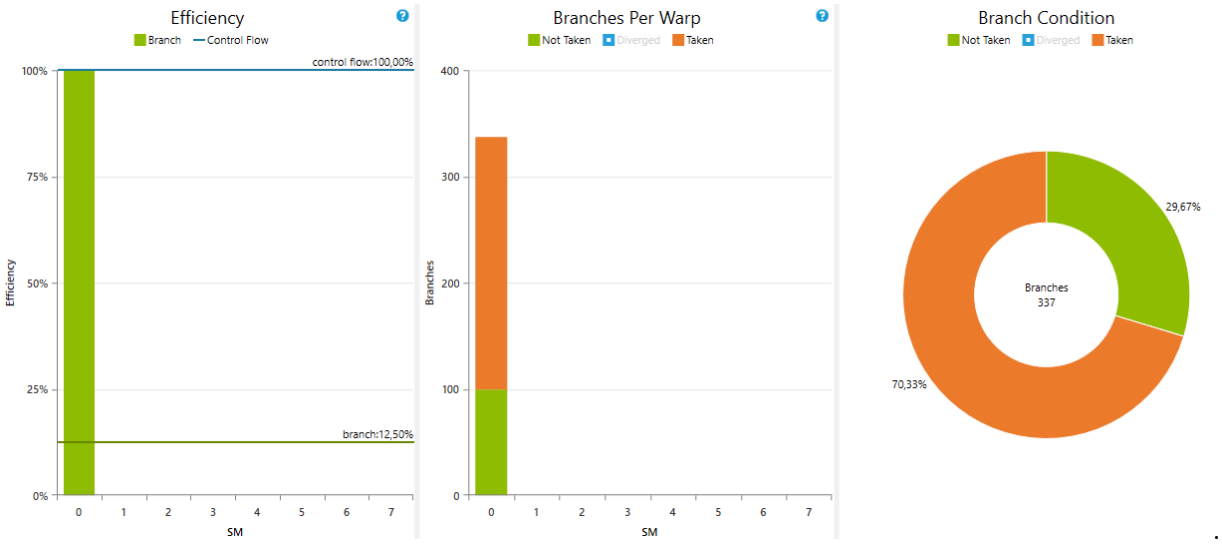

Figure 4.68: Branch statistics of Add function

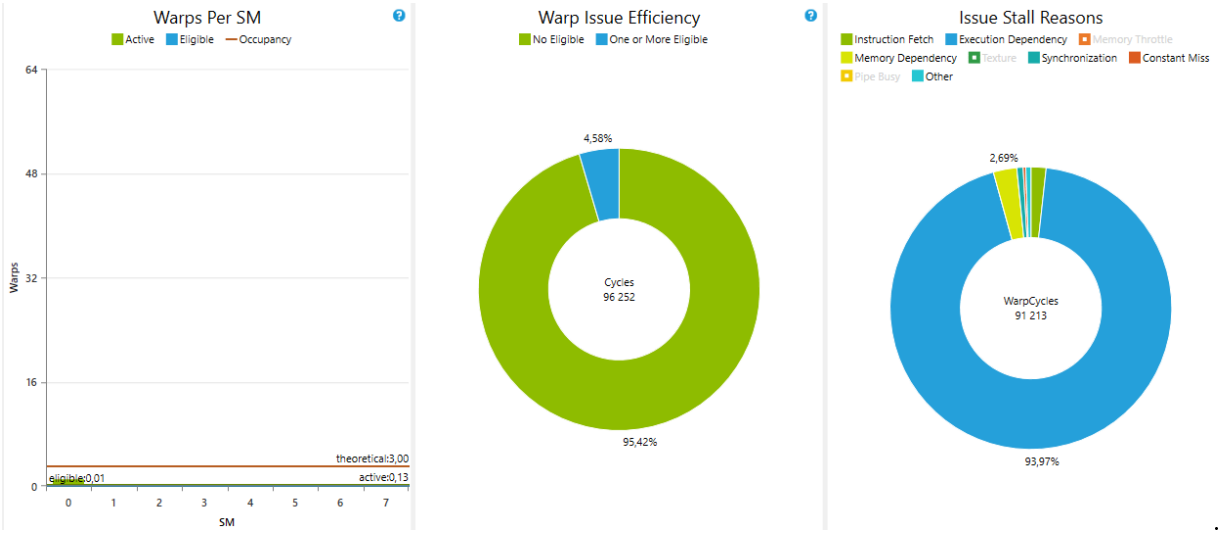

Figure 4.69: Issue efficiency of Add function

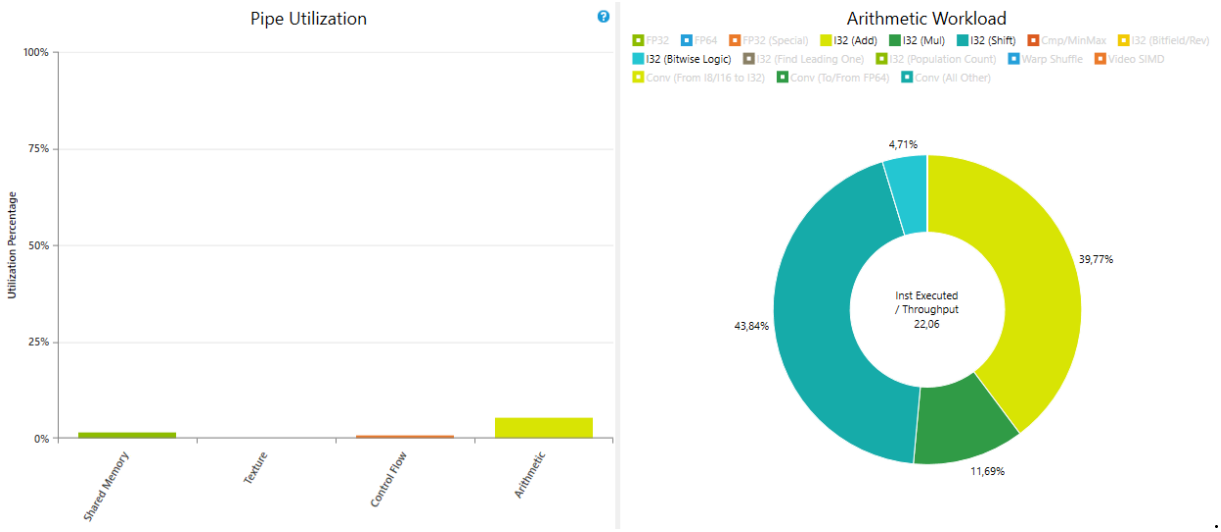

Figure 4.70: Pipe utilization of Add function

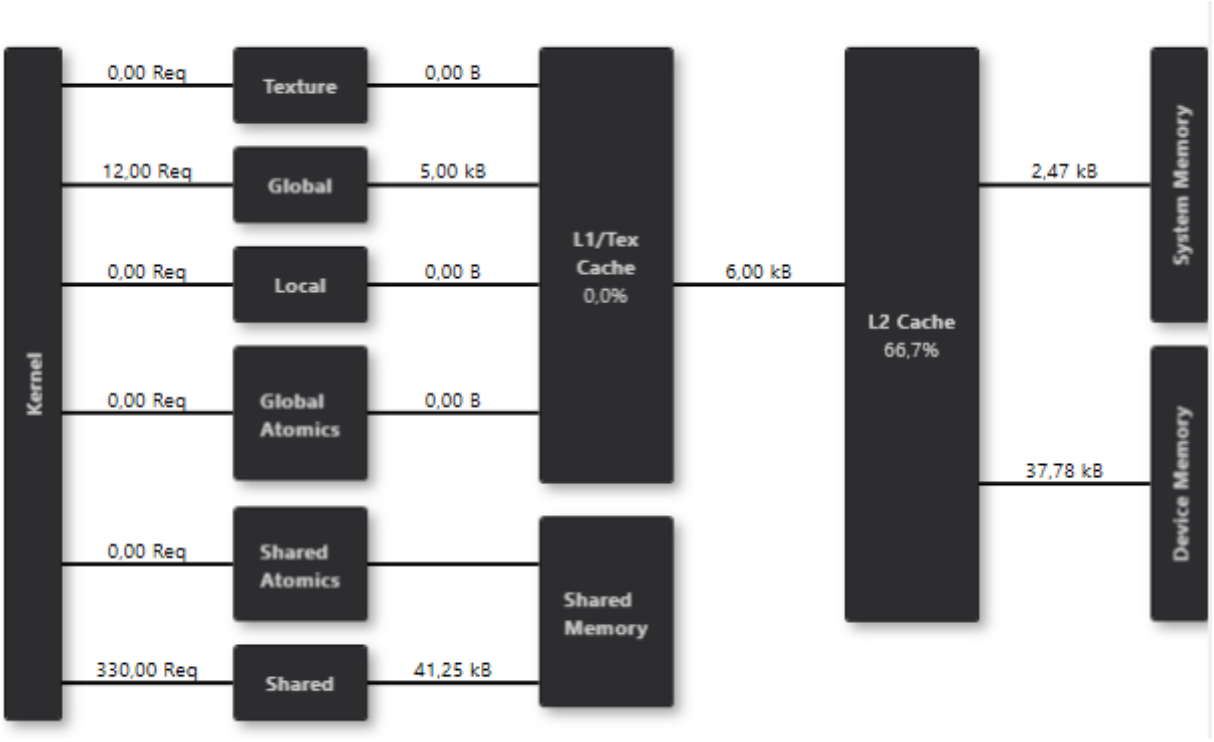

Figure 4.71: Memory overview of Add function

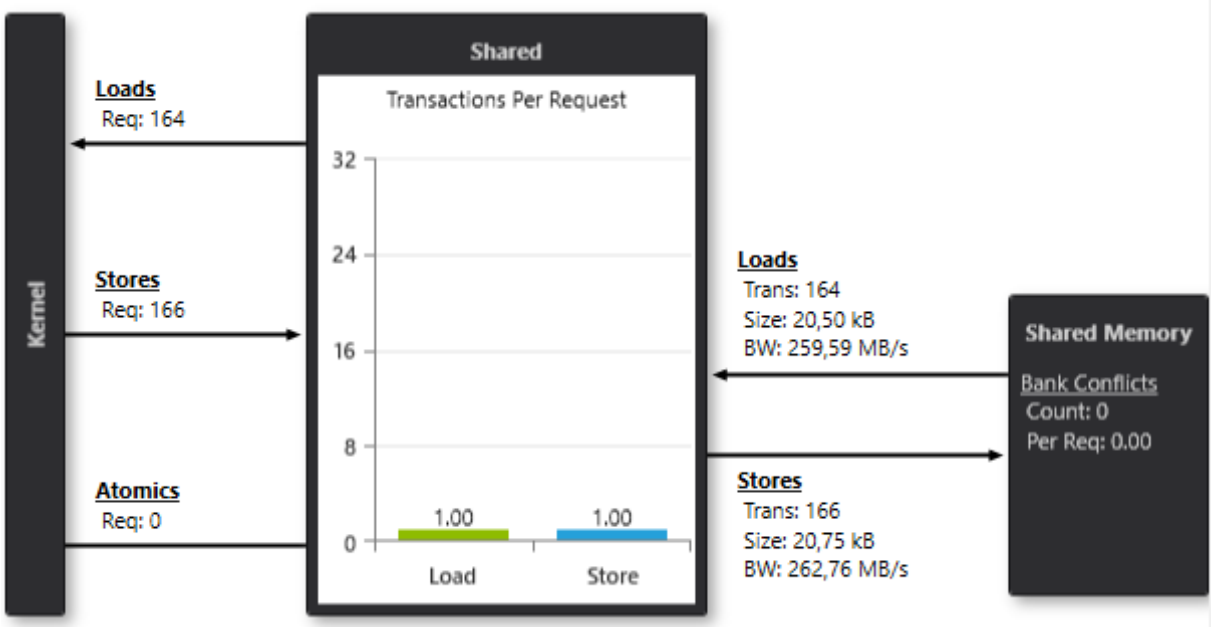

Figure 4.72: Shared memory chart of Add function

4.8 Subtract

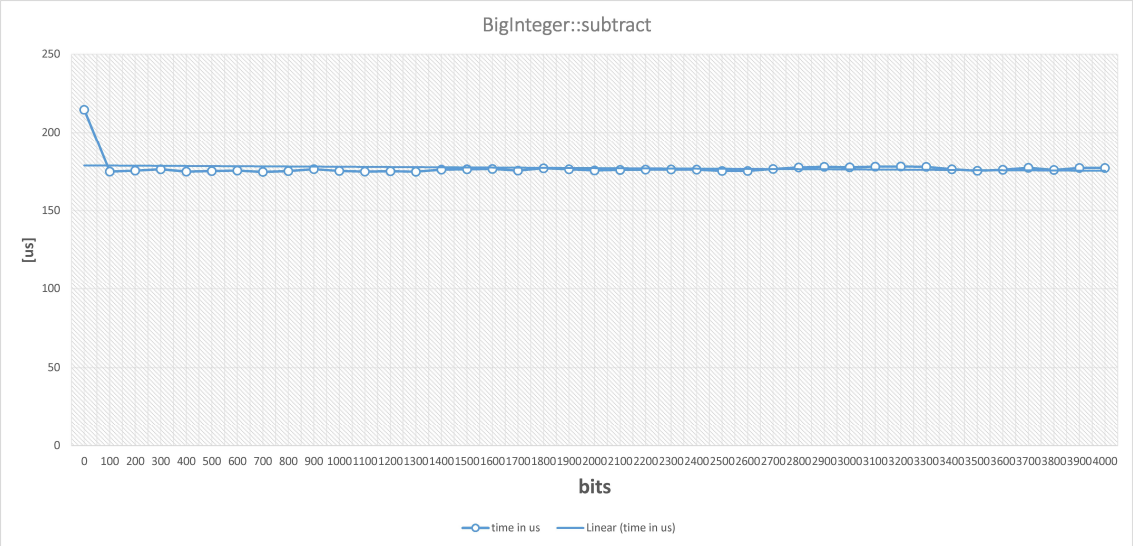

Figure 4.73: Execution time of function Subtract in μs depending on input's bitwise length

The implementation of Subtract function is almost identical as Add. Surprisingly the execution time is much more stable. The average time is higher probably because of additional instructions in storing borrow value. The statistics look the same.

Variable	Achieved	Theoretical	Device Limit	
Occupancy Per SM				
Active Blocks		3	32	0 10 20 30
Active Warps	0.12	3	64	0 20 40 60
Active Threads		96	2048	0 1000 2000
Occupancy	0.20 %	4.69 %	100.00 %	0% 50% 100%
Warps				
Threads/Block		32	1024	0 500 1000
Warps/Block		1	32	0 10 20 30
Block Limit		64	32	0 10 20 30
Registers				
Registers/Thread		14	255	0 100 200
Registers/Block		512	65536	0 20000 40000 60000
Registers/SM		1536	65536	0 20000 40000 60000
Block Limit		128	32	0 10 20 30
Shared Memory				
Shared Memory/Block		27280	49152	0 20000 40000
Shared Memory/SM		81840	98304	0 50000
Block Limit		3	32	0 10 20 30

Figure 4.74: Occupancy statistics of Subtract function

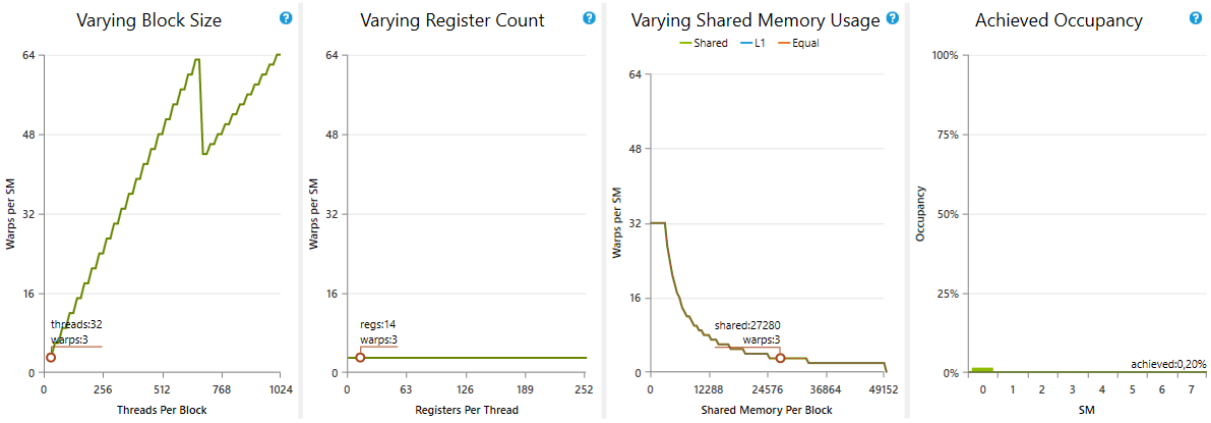

Figure 4.75: Occupancy charts of Subtract function

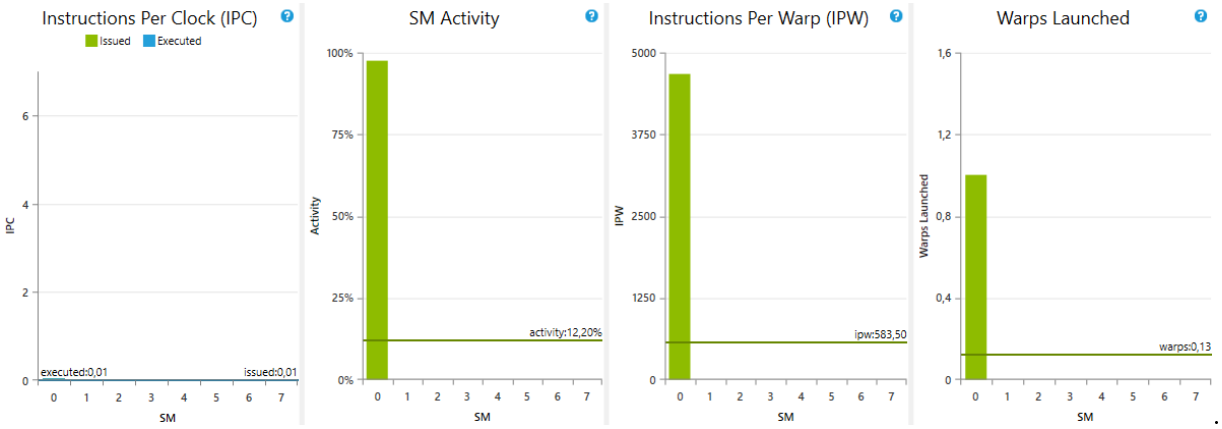

Figure 4.76: Instruction statistics of Subtract function

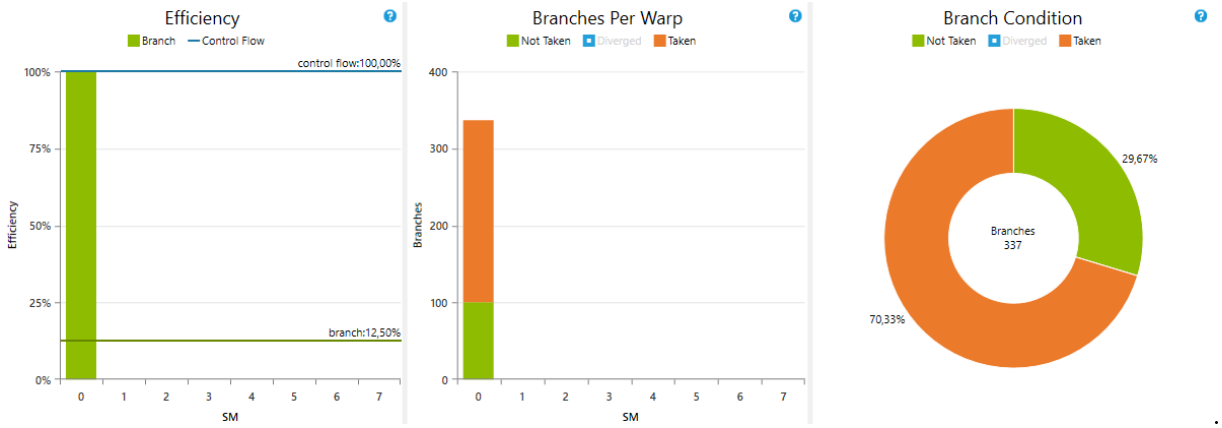

Figure 4.77: Branch statistics of Subtract function

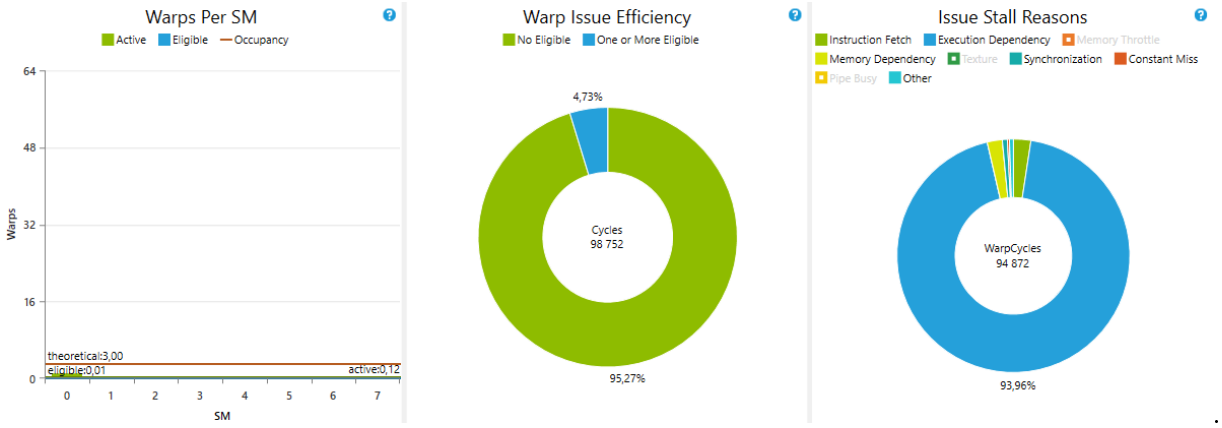

Figure 4.78: Issue efficiency of Subtract function

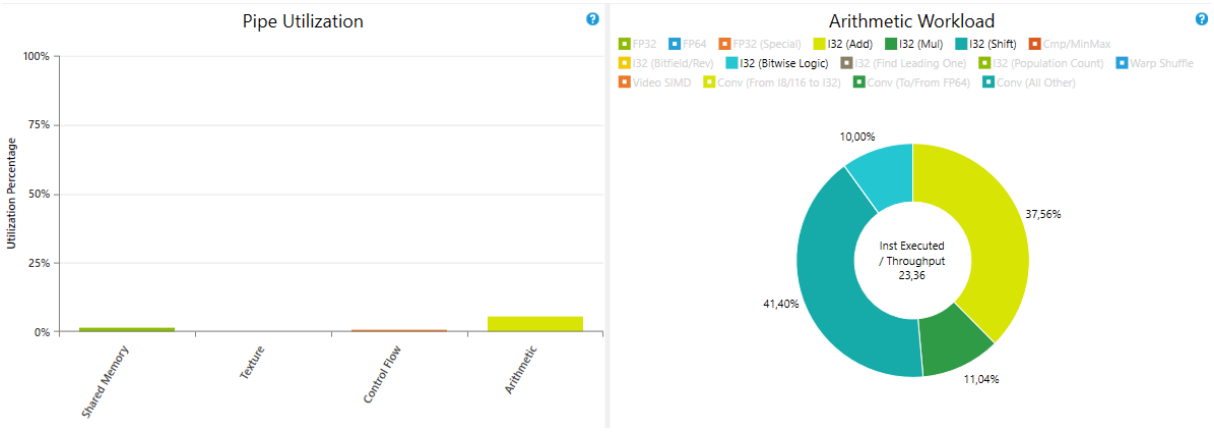

Figure 4.79: Pipe utilization of Subtract function

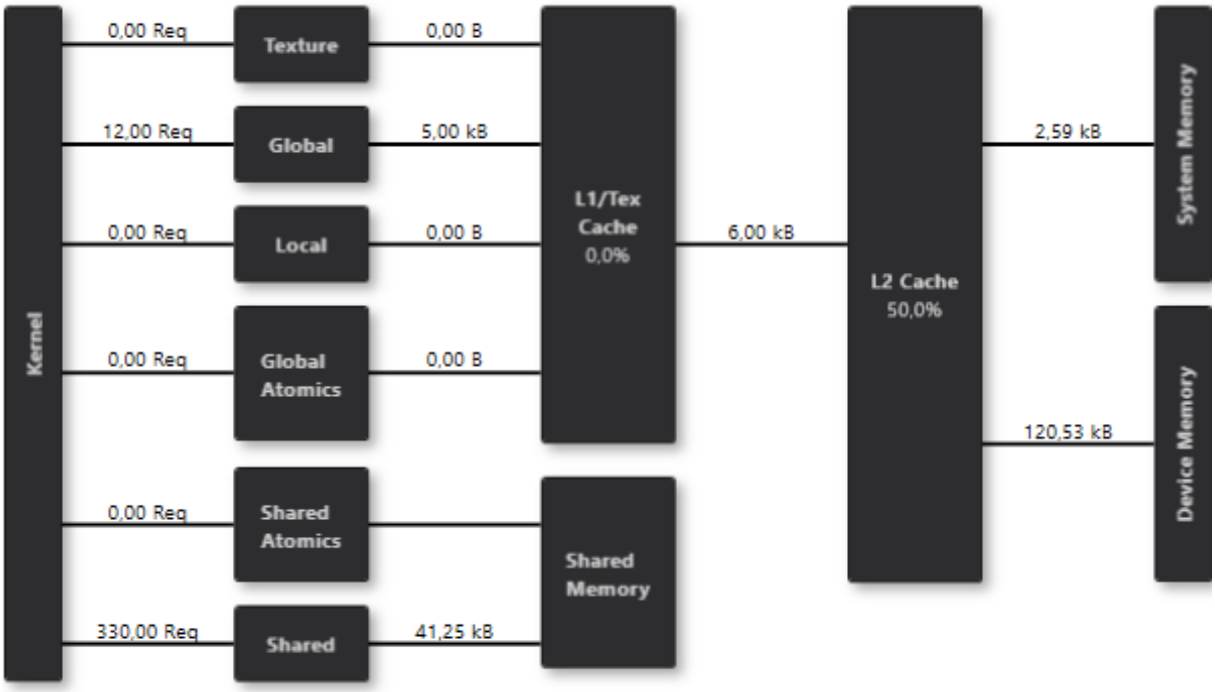

Figure 4.80: Memory overview of Subtract function

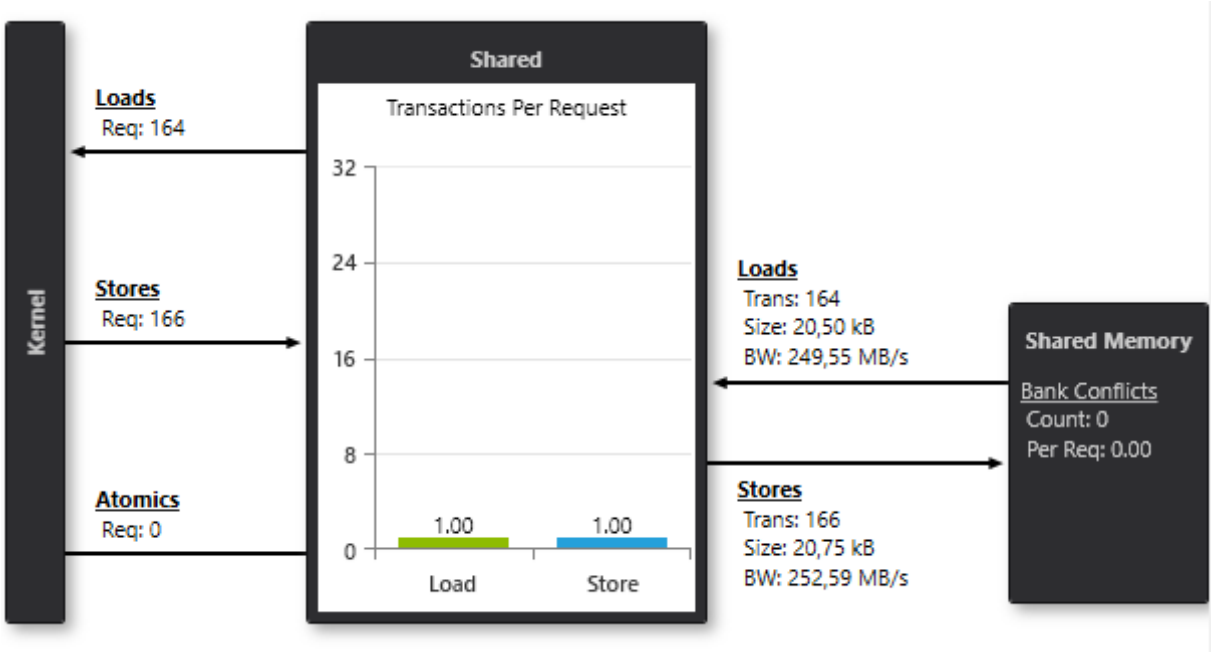

Figure 4.81: Shared memory chart of Subtract function

4.9 Multiply

The multiply initial configuration is 4 blocks, 64 threads each. Occupancy is limited to 6.25% by shared memory. The computations are evenly distributed between 4 SMs.

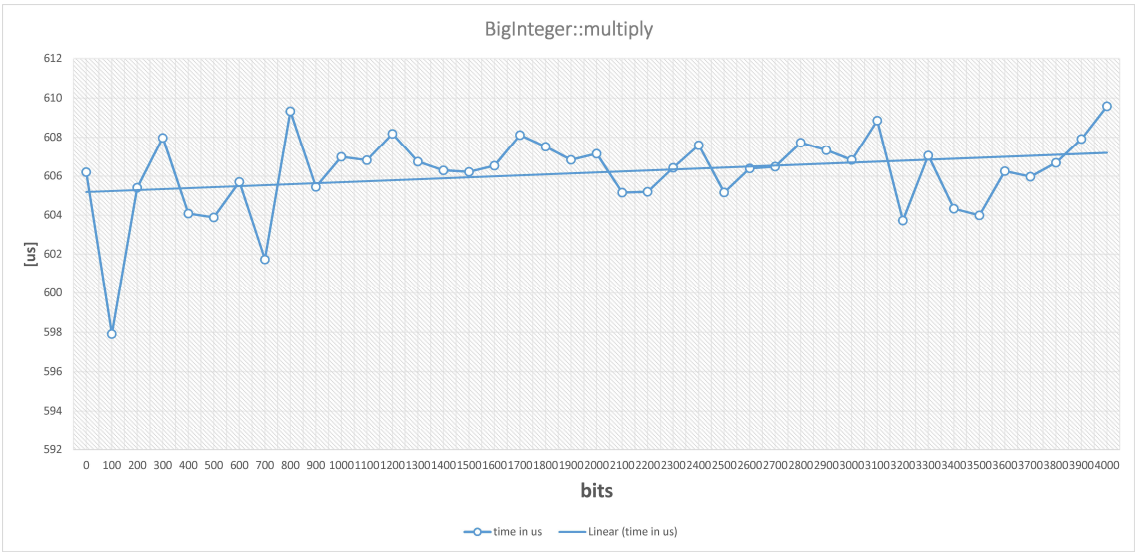

Figure 4.82: Execution time of function Multiply in μs depending on input's bitwise length

Variable	Achieved	Theoretical	Device Limit	
Occupancy Per SM				
Active Blocks		2	32	0 10 20 30
Active Warps	0.77	4	64	0 20 40 60
Active Threads		128	2048	0 1000 2000
Occupancy	1.20 %	6.25 %	100.00 %	0% 50% 100%
Warps				
Threads/Block		64	1024	0 500 1000
Warps/Block		2	32	0 10 20 30
Block Limit		32	32	0 10 20 30
Registers				
Registers/Thread		21	255	0 100 200
Registers/Block		1536	65536	0 20000 40000 60000
Registers/SM		3072	65536	0 20000 40000 60000
Block Limit		42	32	0 10 20 30
Shared Memory				
Shared Memory/Block		44176	49152	0 20000 40000
Shared Memory/SM		88352	98304	0 50000
Block Limit		2	32	0 10 20 30

Figure 4.83: Occupancy statistics of Multiply function

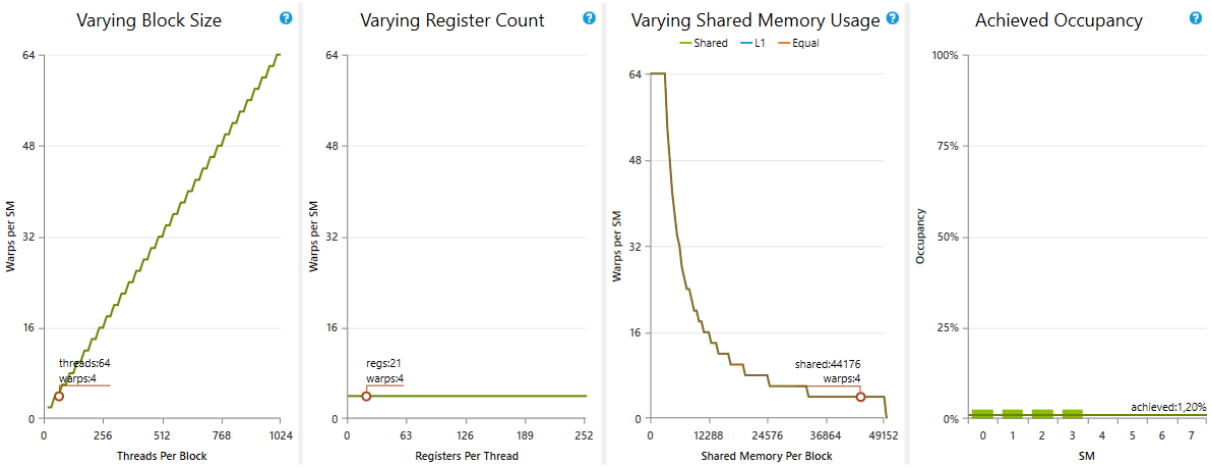

Figure 4.84: Occupancy charts of Multiply function

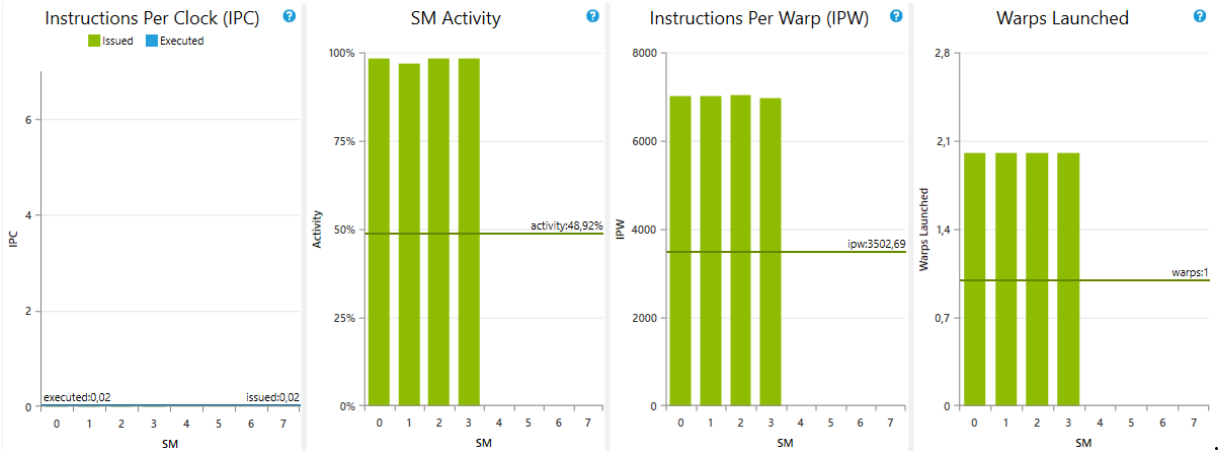

Figure 4.85: Instruction statistics of Multiply function

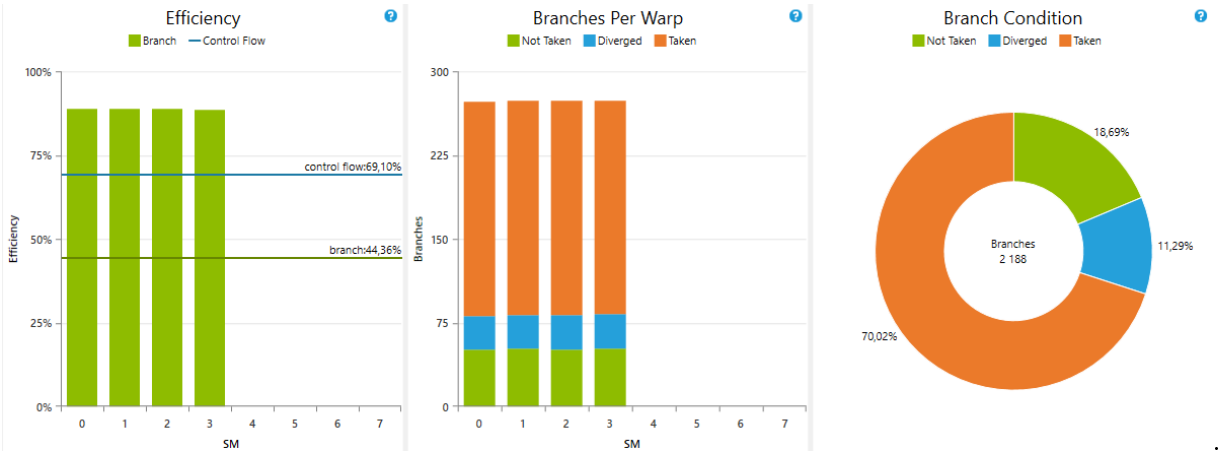

Figure 4.86: Branch statistics of Multiply function

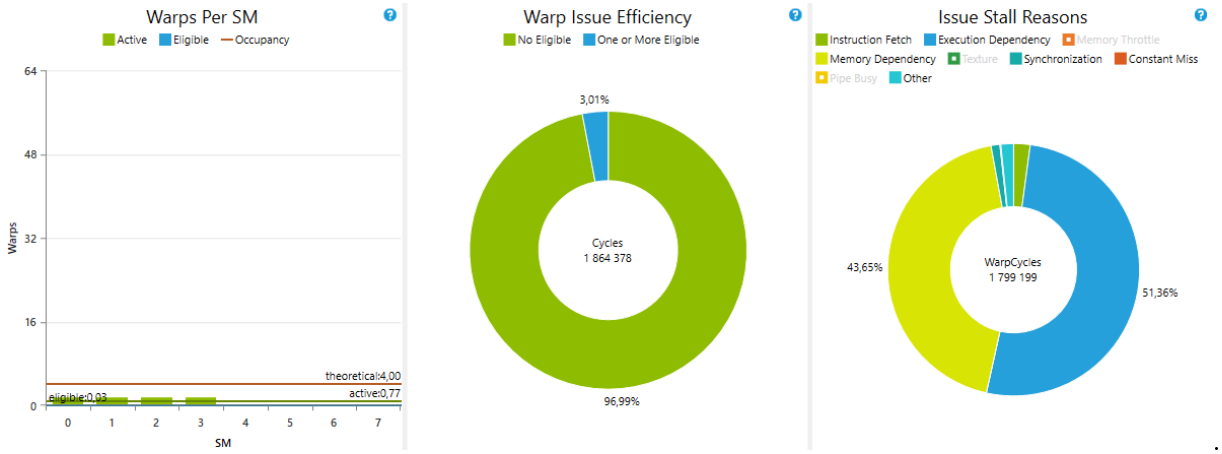

Figure 4.87: Issue efficiency of Multiply function

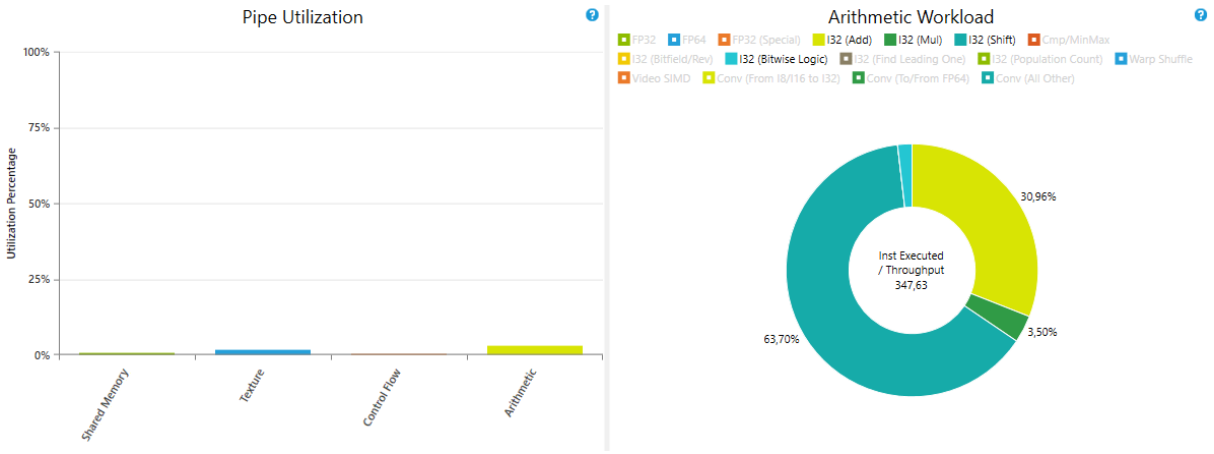

Figure 4.88: Pipe utilization of Multiply function

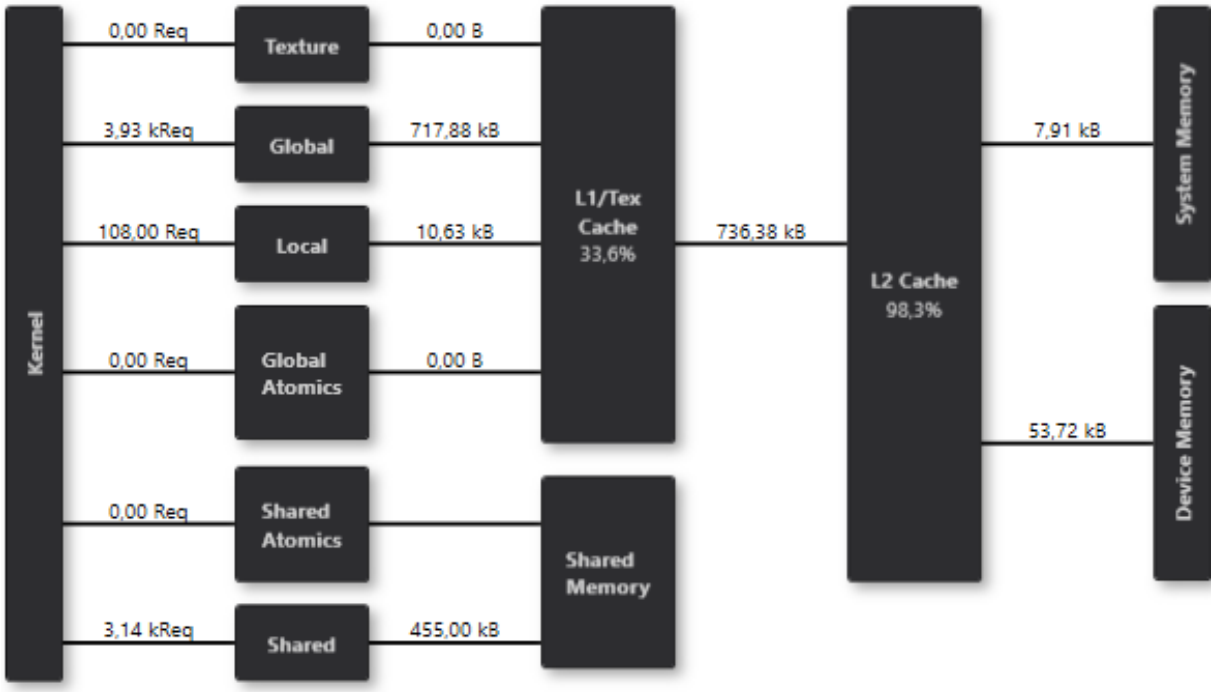

Figure 4.89: Memory overview of Multiply function

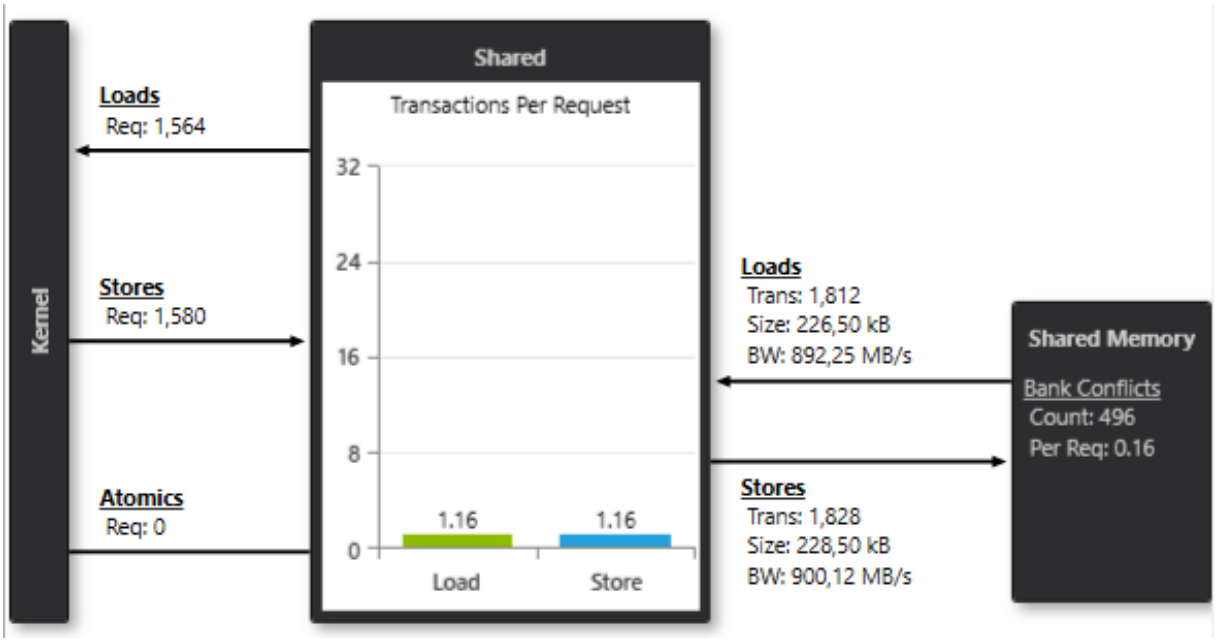

Figure 4.90: Shared memory chart of Multiply function

4.10 Modulo reduction

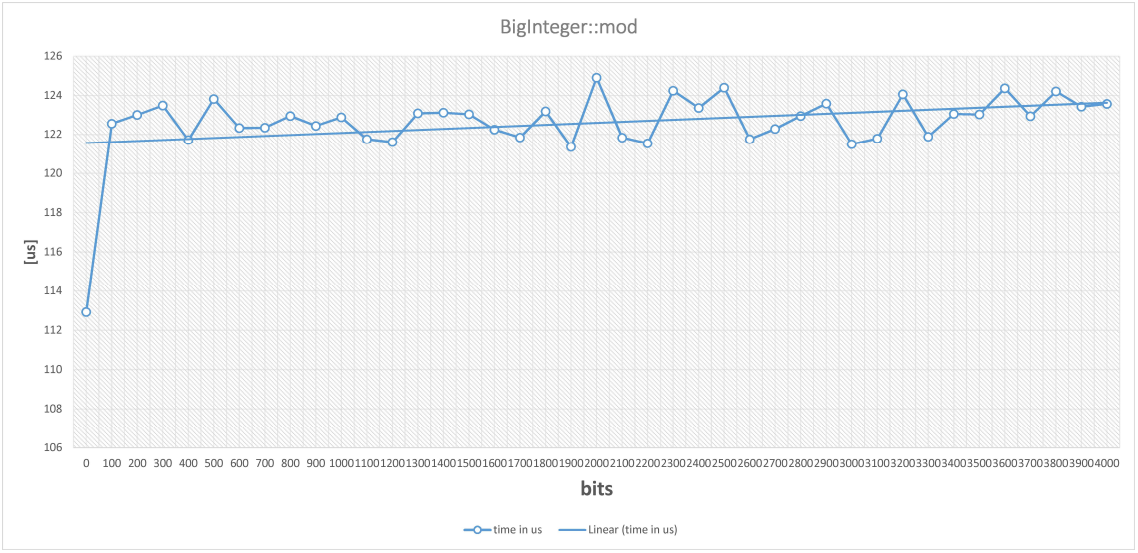

Figure 4.91: Execution time of function Modulo reduction in μs depending on input's bitwise length

The graph4.91 presents measurements of operation X mod M, where bitwise lengths of both X and M increase, and the length of X i always one bit less the M. During tests four blocks were launched, each with different values. The chart4.95 shows how higher values on SM 0 block activity of other SMs. This is caused by use of "while" loop in implementation. For higher values the loop needs to makes more rounds which needs more time for computation, eventually decreasing performance.

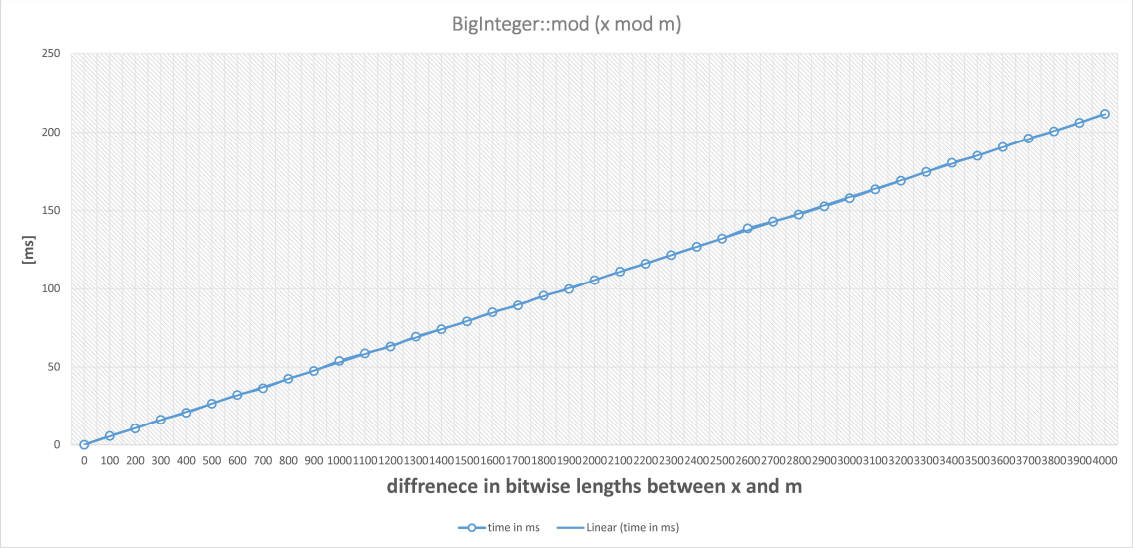

Figure 4.92: Execution time of function Modulo reduction in ms depending on inputs' bitwise lengths differences

The graph above shows how the function performance depends on the difference in lengths of X and M. Modulus M is set at 8 bits, whereas bitwise length of value X increases. This causes more repetitions of the loop which affects performance.

Variable	Achieved	Theoretical	Device Limit	
Occupancy Per SM				
Active Blocks		3	32	0 10 20 30
Active Warps	0.50	12	64	0 20 40 60
Active Threads		384	2048	0 1000 2000
Occupancy	0.78 %	18.75 %	100.00 %	0% 50% 100%
Warps				
Threads/Block		128	1024	0 500 1000
Warps/Block		4	32	0 10 20 30
Block Limit		16	32	0 10 20 30
Registers				
Registers/Thread		23	255	0 100 200
Registers/Block		3072	65536	0 20000 40000 60000
Registers/SM		9216	65536	0 20000 40000 60000
Block Limit		21	32	0 10 20 30
Shared Memory				
Shared Memory/Block		27280	49152	0 20000 40000
Shared Memory/SM		81840	98304	0 50000
Block Limit		3	32	0 10 20 30

Figure 4.93: Occupancy statistics of Modulo reduction function

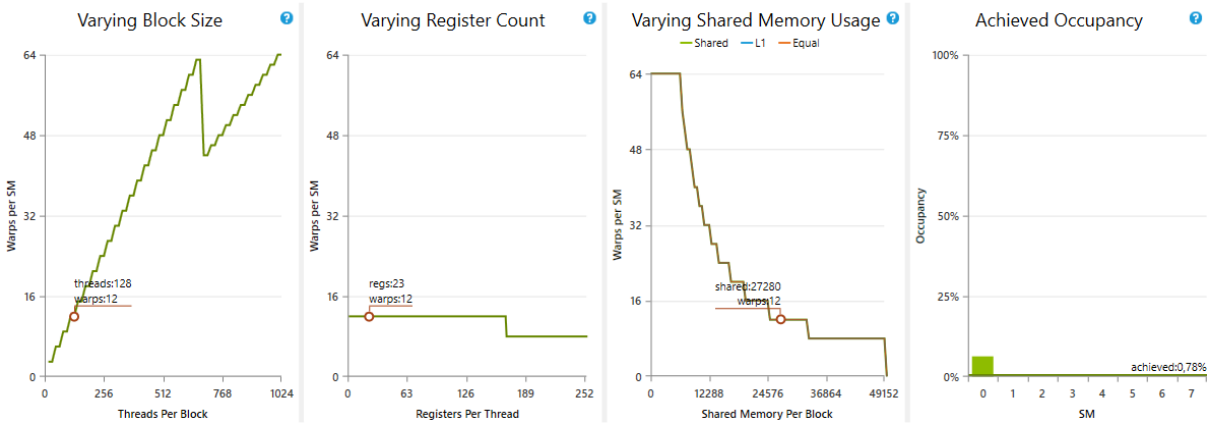

Figure 4.94: Occupancy charts of Modulo reduction function

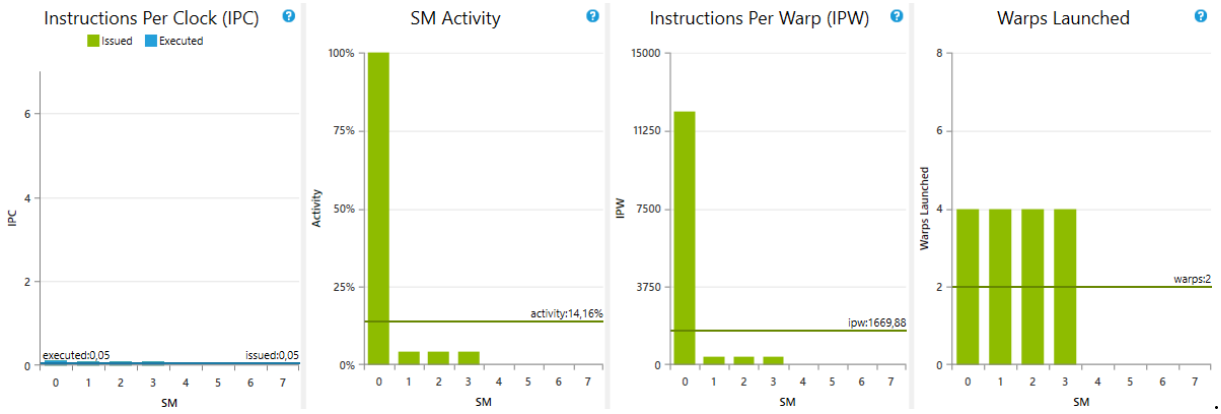

Figure 4.95: Instruction statistics of Modulo reduction function

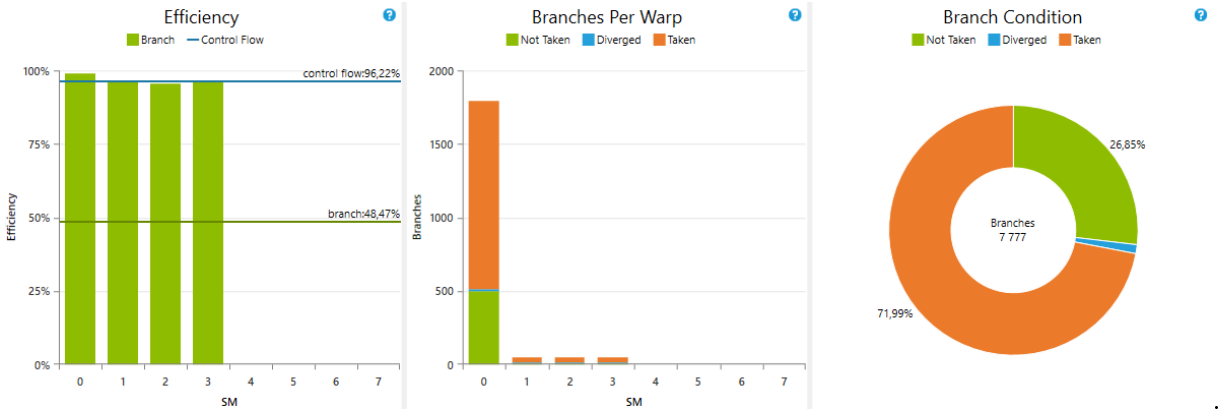

Figure 4.96: Branch statistics of Modulo reduction function

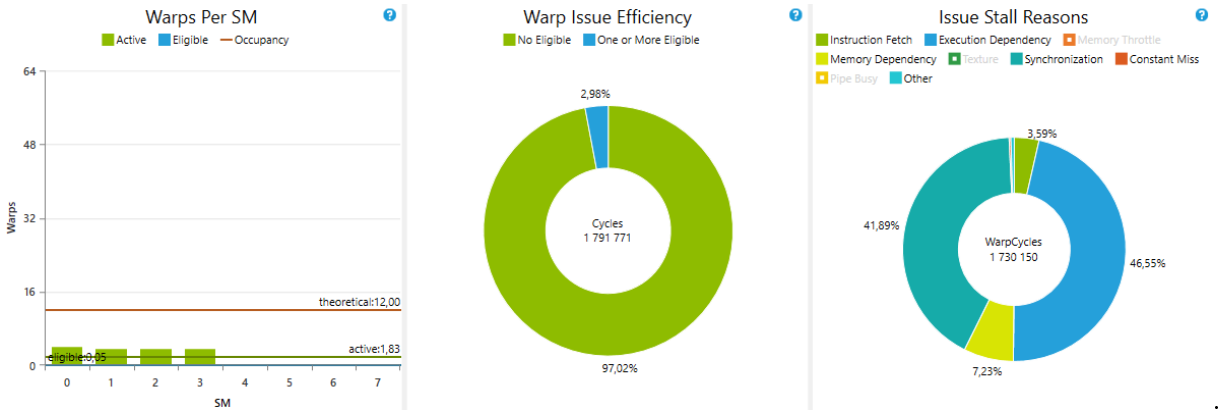

Figure 4.97: Issue efficiency of Modulo reduction function

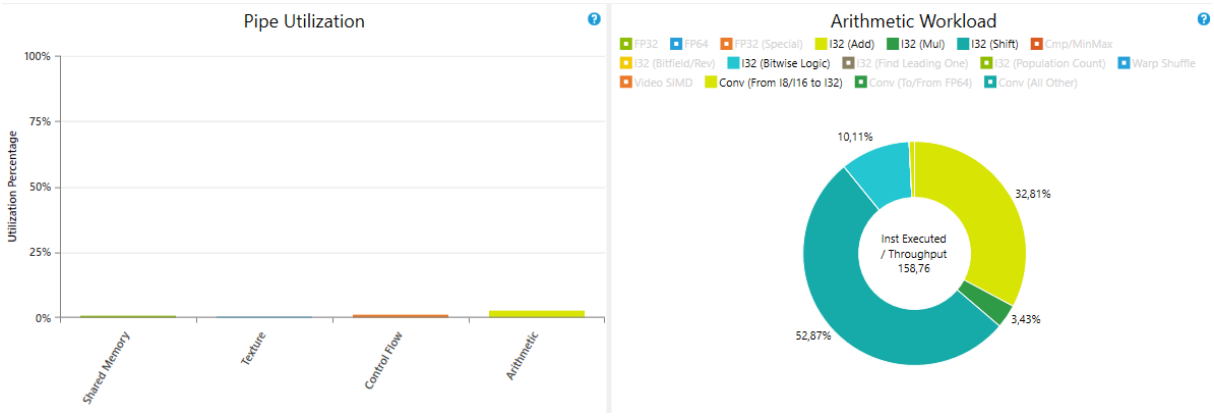

Figure 4.98: Pipe utilization of Modulo reduction function

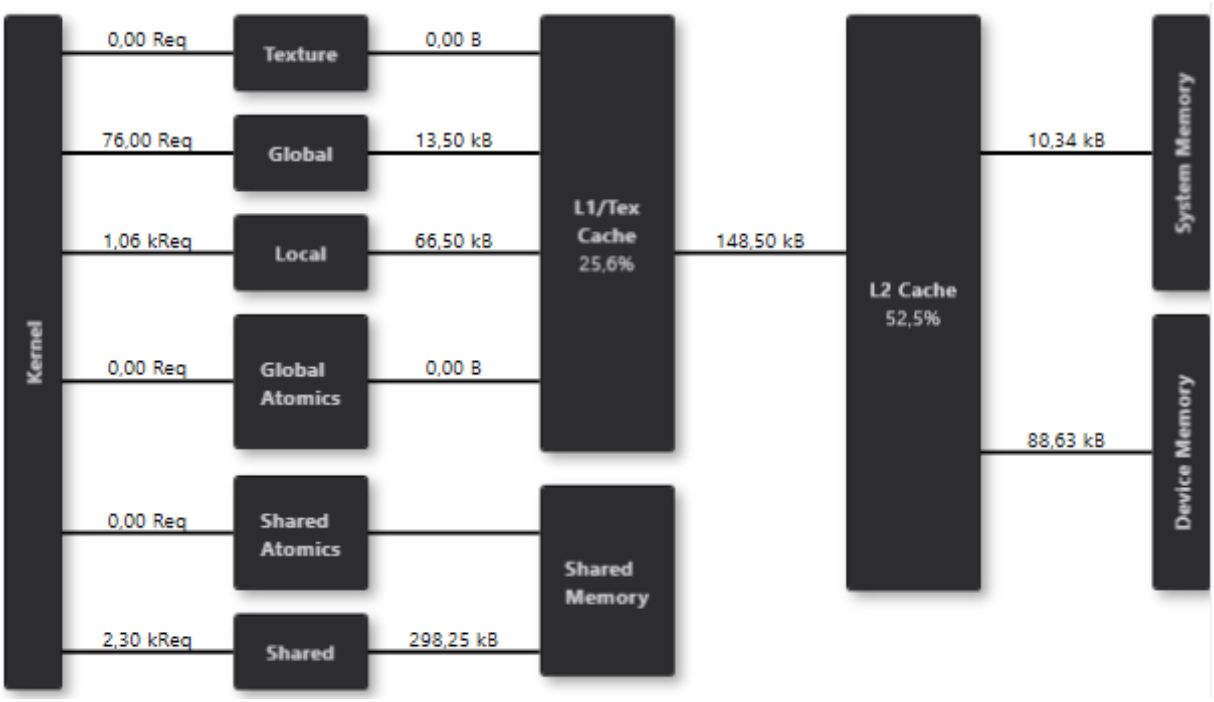

Figure 4.99: Memory overview of Modulo reduction function

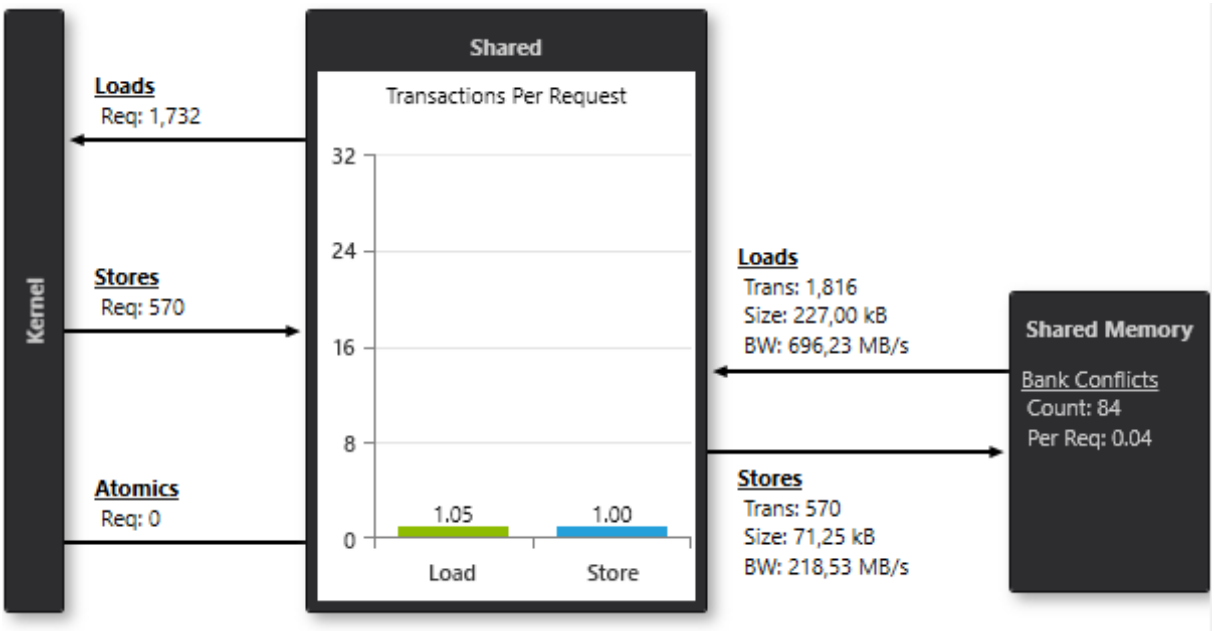

Figure 4.100: Shared memory chart of Modulo reduction function

4.11 Multiply modulo

The function multiply modulo composes of multiply function and couple of calls to modulo reduction. For constant difference in lengths between multiplication factors and modulus the function operates stable.

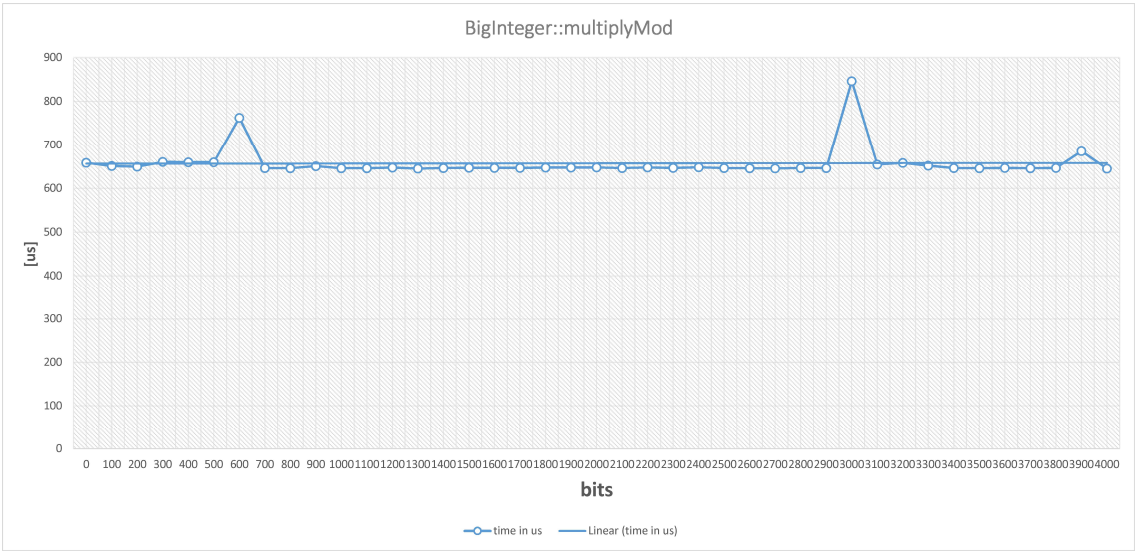

Figure 4.101: Execution time of function MultiplyMod in μs depending on input's bitwise length

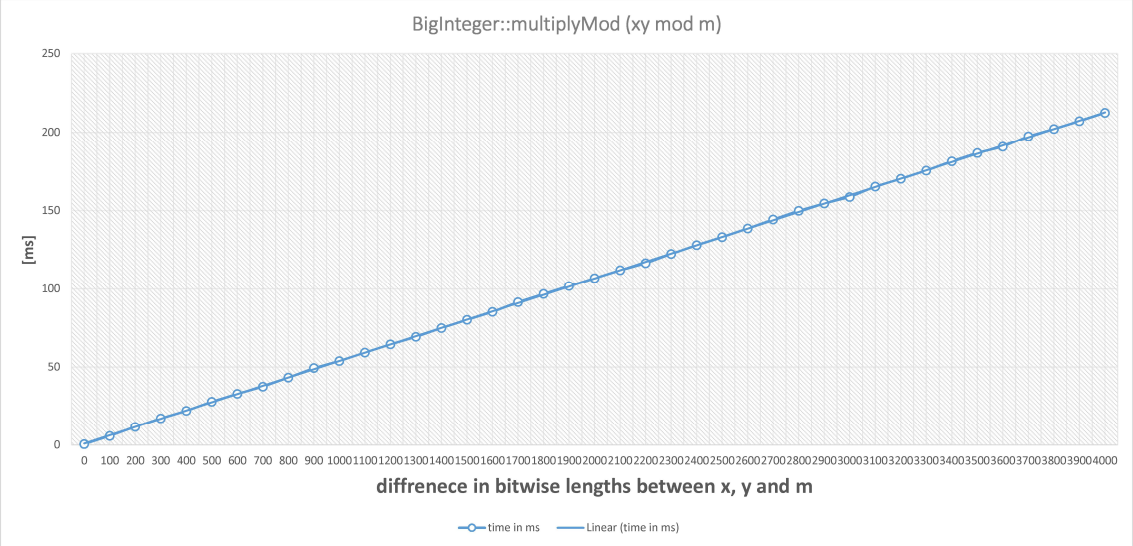

Figure 4.102: Execution time of function MultiplyMod in ms depending on inputs' bitwise lengths differences

The chart above shows how overhead of modulo reductions function seizes the final performance.

4.12 Power modulo

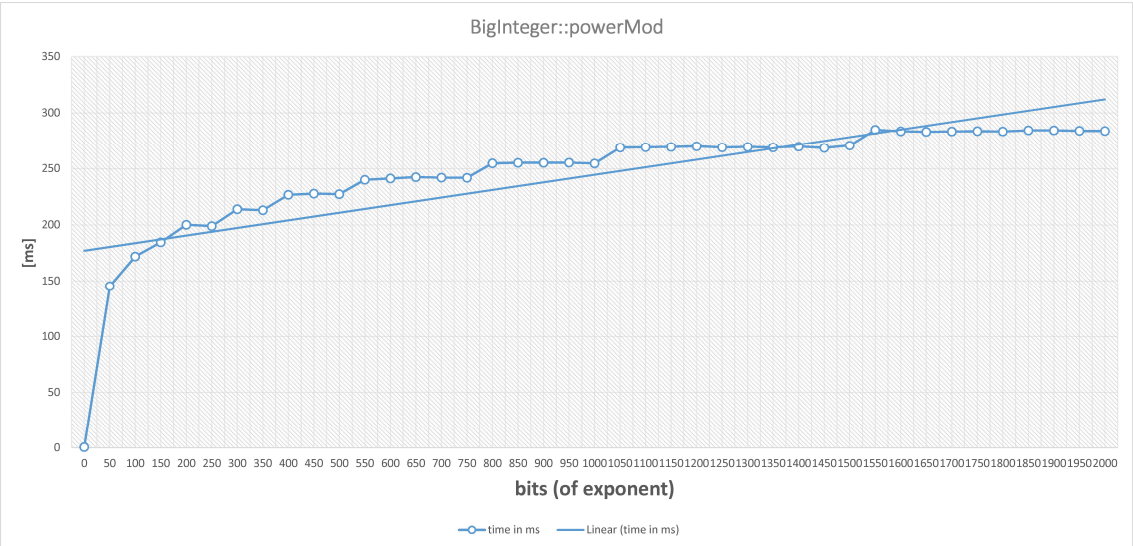

Figure 4.103: Execution time of function Power modulo in ms depending on exponent's bitwise length

The modular exponentiation was implemented using Montgomery Ladder. The complexity of the algorithim is:

$$log_2(bits)$$

which is clearly presented on the chart characteristic.4.103

4.13 RSA encrypt

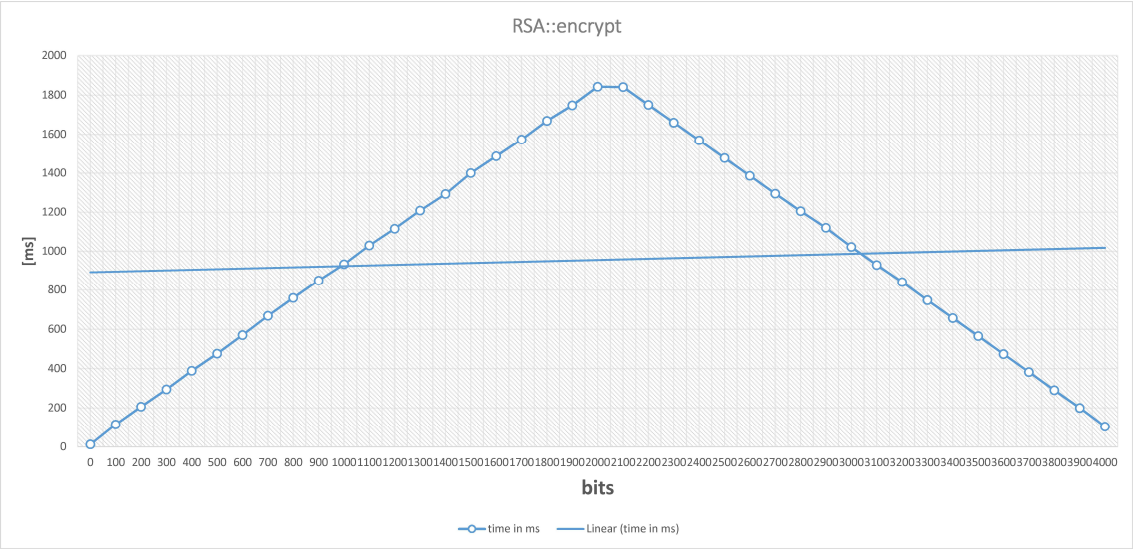

Figure 4.104: Execution time of function encrypt in ms depending on message's bitwise length

RSA encryption uses modular exponentiation with exponent set to 65537. As the message's length rises so does the encryption time. This is caused again by modular reduction. At 2000 bits threshold the values begin to overflow the magnitude array which causes undefined behavior. This also shows the correctness of calculations stops at 2K.

Chapter 5

Conclusions

This project approaches the general problems concerning implementations of cryptographic schemes. Following from that, it mainly focuses on multiple precision integer arithmetics and solutions using parallelized computing. Large numbers are the keystone in security. However, performance and efficiency become collateral when side-channel attacks are taken into the account. Even the most secure cryptographic systems become vulnerable to penetration and exploitable when improperly implemented.

Successful parts of this project are definitely the implementations of BigInteger library and DeviceWrapper class, providing a very good base for further development and improvement. Most of the implemented functions, like add, subtract, multiply, shift left, shift right, compare, equals, clear and clone, are working efficiently and are resistant to timing attacks. Some parts of the code would certainly need to be rewritten, especially modular reduction algorithm, if the project was to fully meet imposed assumptions. In chapter 4, it was shown how much more work can be done to escalate parallelization and to provide genuine resistance to more side-channel attacks.

CUDA technology is a growing architecture which constantly provides new tools to harness the GPUs true potential. It has been a solution to many projects restrained by weak computational performance, and nowadays it becomes a standard in programming cryptographic systems. The platform contains much more features which could freely be implemented within this application. (e.g. dynamic parallelism[6])

Further work will surly require much more analysis, testing and adjustments to bring this project to usable state. Many loose ends prompt constant development, as well as never-stopping requisition for fast and secure cryptographic systems.

Bibliography

[1] S. Baktr, E. Savas. Highly-parallel montgomery multiplication for multi-core general-purpose microprocessors. https://eprint.iacr.org/2012/140.pdf. Available: 2017-08-28.

[2] I. Buck. High performance computing with cuda. https://mc.stanford.edu/cgi-bin/images/b/ba/M02_2.pdf. Available: 2017-08-28.

[3] G. G. Christophe Clavier, Benoit Feix. Square always exponentiation. http://2011.indocrypt.org/slides/verneuil.pdf. Available: 2017-08-28.

[4] S. Cook. Cuda programming a developer's guide to parallel computing with gpus. http://www.hds.bme.hu/~fhegedus/C++/Shane%20Cook%20-%20CUDA%20Programming%20-olvasOM.pdf. Available: 2017-08-28.

[5] N. CORPORATION. Cuda documentation. http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#axzz4rFBJxFp0. Available: 2017-08-28.

[6] N. CORPORATION. Dynamic parallelism in cuda. http://developer.download.nvidia.com/assets/cuda/files/CUDADownloads/TechBrief_Dynamic_Parallelism_in_CUDA.pdf. Available: 2017-08-28.

[7] N. CORPORATION. Nvidia nsight visual studio edition 5.3 user guide. http://docs.nvidia.com/nsight-visual-studio-edition/5.3/Nsight_Visual_Studio_Edition_User_Guide.htm#Nsight_Visual_Studio_Edition_User_Guide.htm%3FTocPath%3D_____1. Available: 2017-08-28.

[8] Y. Ge. A note on the carmichael function. http://yimin-ge.com/doc/carmichael.pdf. Available: 2017-08-28.

[9] E. O. Iskra Nunez. Generalized hamming weights for linear codes. http://www.uprh.edu/~simu/Reports2001/NOU.pdf, 2001. Available: 2017-08-28.

[10] L. I. Jean-Claude Bajard, G. A. Jullien. Parallel montgomery multiplication in gf(2k) using trinomial residue arithmetic. https://eprint.iacr.org/2012/140.pdf. Available: 2017-08-28.

[11] M. E. Kaihara. An implementation of rsa 2048 on gpus using cuda. http://www.marcelokaihara.com/papers/An_Implementation_of_RSA2048_on_GPUs_using_CUDA.pdf. Available: 2017-08-28.

[12] R. Laboratories. Rsaes-oaep encryption scheme algorithm specification and supporting documentation. http://www.inf.pucrs.br/~calazans/graduate/TPVLSI_I/RSA-oaep_spec.pdf. Available: 2017-08-28.

[13] D. Luebke. Scalable parallel programming for high-performance scientific computing. `http://vision.lbl.gov/People/han/isbi2008/pdfs/0000836.pdf`. Available: 2017-08-28.

[14] Microsoft. Visual stdio. `https://www.visualstudio.com`. Available: 2017-08-28.

[15] E. Milanov. The rsa algorithm. `https://sites.math.washington.edu/~morrow/336_09/papers/Yevgeny.pdf`, June 2009. Available: 2017-08-27.

[16] NVIDIA. Cuda zone. `https://developer.nvidia.com/cuda-zone`. Available: 2017-08-27.

[17] NVIDIA. Using inline ptx assembly in cuda. `https://www.cs.cmu.edu/afs/cs/academic/class/15668-s11/www/cuda-doc/ptx_isa_2.2.pdf`. Available: 2017-08-28.

[18] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krüger, A. Lefohn, T. J. Purcell. A survey of general-purpose computation on graphics hardware. *Computer Graphics Forum*, 26(1):80–113, 2007.

[19] P. H. G. Schaathun. Square and multiply introducing rsa. `http://www.hg.schaathun.net/DisMath/PK04/04squaremultiply.pdf`. Available: 2017-08-28.

[20] Z. C. P. Schaumont. A parallel implementation of montgomery multiplication on multi-core systems: Algorithm, analysis, and prototype. `http://rijndael.ece.vt.edu/chenzm/pSHSTC.pdf`. Available: 2017-08-28.

[21] W. Stein. Elementary number theory: Primes, congruences, and secrets. `http://wstein.org/ent/ent.pdf`, 2017. Available: 2017-08-28.

[22] D. F. YongBin Zhou. Side-channel attacks: Ten years after its publication and the impacts on cryptographic module security testing. `https://eprint.iacr.org/2005/388.pdf`. Available: 2017-08-28.

List of Figures

List of Tables