IS2202 - Laboration 2
Simulating Graphics Processing Units (GPUs) in GPGPU-Sim

Linghan Zhao   Teammate: Han Tu

March 11, 2024

In the preparation part, we install Ubuntu 22.04 natively as a dual boot and VirtualBox.

This lab is to learn how simulators can be used to perform exploration on Graphics Processing Units (GPUs), using the GPGPU-Sim simulator.

Exercise 1: Compiling CUDA applications (Directed part)

Our first exercise is to compile CUDA applications, simulating them using GPGPU-Sim.

Question 1

The CUDA kernel is called matrix_multiply. The signature of the kernel is: __global__ void matrix_multiply(float *A, float *B, float* C, int N). The data type is floating.

Question 2

For each result in C. it needs 2*N Floating Point Operations(N multiplication and N addition). And C has N*N elements in total. So 2*N*N*N Floating Point Operations are needed in total.

Exercise 2: Simulating Matrix Multiplication on the GTX-480

Question 3

Look up the NVIDIA GTX480 GPU and describe its characteristics.

NVIDIA GTX480 GPU	Release Year: March 26th, 2010
Peak performance	Pixel Rate : 21.03 GPixel/s
	Texture Rate : 42.06 GTexel/s
	FP32 (float) : 1,345 GFLOPS
	FP64 (double) : 168.1 GFLOPS (1:8)
External memory bandwidth	177.4 GB/s
Architecture family	Fermi
Core Frequency	GPU Clock : 701 MHz
	Shader Clock : 1401 MHz
	Memory Clock : 924 MHz 3.7 Gbps effective

Table 1: NVIDIA GTX480 GPU characteristics

Question 4

When compute a fairly small matrix multiplication of size 32x32, the simulation takes 1.690 s.

The GPGPU simulator gives a report shows that "gpu_sim_cycle = 5368", and the frequency is 701 MHz, so we can calculate the estimated time for a real GPU

$$\begin{aligned} estimated\ real\ CPU\ time &= clock\ cycles \times \frac{1}{frequency} &(1)\\ &= \frac{5368}{701 \times 10^6} &(2)\\ &= 7.66 \times 10^{-6} s &(3)\\ slow\ down &= \frac{simulation\ time}{real\ GPU\ time} &(4)\\ &= \frac{1.690}{7.66 \times 10^{-6}} &(5)\\ &= 2.2 \times 10^5 &(6) \end{aligned}$$

The slowdown is so large compared to executing on a real GPU, the reasons might be

1.Hardware Emulation Overhead: GPGPU simulators often need to emulate the behavior of GPU hardware on a traditional CPU, which introduces overhead due to the fundamental differences in architecture and execution paradigms.

2.Instruction Translation: Simulators typically translate GPU instructions into CPU instructions, which adds additional processing time and overhead.

3.Parallelism: GPUs excel at executing parallel tasks efficiently, leveraging thousands of cores simultaneously. Simulators may struggle to fully utilize the available CPU cores or to replicate the level of parallelism found in real GPU hardware.

There are other reasons effect, like simulator may be lack of special optimization, GPU resource Sharing and scheduling, and the memory access patterns.

Question 5

Analyze stat.txt and answer the following questions:

a) It takes *gpu_sim_cycle* = 5368 clock cycles to execute the GEMM kernel.

b) *gpu_sim_insn* = 154624 instructions were executed.

c) The GPU achieved $gpu\_ipc = 28.8048$ (instructions-per-cycle).

d) There's a formular to calculate the DRAM Bandwidth,

$$
\begin{aligned}
&Peak\ off-chip\ DRAM\ bandwidth = gpgpu\_n\_mem* && (7)\\
&gpgpu\_n\_mem\_per\_ctrlr * gpgpu\_dram\_buswidth * DRAM\ Clock * 2(forDDR) && (8)\\
&= 6 * 2 * 4 * 924 * 2 && (9)\\
&= 88.704\ GBHz && (10)
\end{aligned}
$$

From the report, we can get "L2_BW = 1.7776 GB/Sec".

- gpgpu_n_mem = Number of memory channels in the GPU (each memory channel has an independent controller for DRAM command scheduling)
- gpgpu_n_mem_per_ctrlr = Number of DRAM chips attached to a memory channel (default = 2, for 64-bit memory channel)
- gpgpu_dram_buswidth = Bus width of each DRAM chip (default = 32-bit = 4 bytes)
- DRAM Clock = the real clock of the DRAM chip (as opposed to the effective clock used in marketing

To calculate L2 cache hit rate, we get these statics from report,
"L2_total_cache_accesses = 426"
"L2_total_cache_misses = 170"
"L2_total_cache_miss_rate = 0.3991"

$$MissRate = \frac{CacheMisses}{CacheMisses + CacheHits} \tag{11}$$

$$HitRate = 1 - MissRate \tag{12}$$

According to the formula, we get the L2 cache hit rate = 1-0.3991=0.6009.

e) It needs 2*N Floating Point Operations(N multiplication and N addition). And it has N*N elements in total. So 2*N*N Floating Point Operations are needed in total.

With the parameter N=32,

$$FLOPS = 2 \times N \times N \times N = 32768 \tag{13}$$

The performance in terms of #FLOPS/s should be

$$
\begin{aligned}
\#FLOPS/s &= \frac{FLOPS}{estimated\ real\ GPU\ time} && (14)\\
&= \frac{32768}{7.66 \times 10^{-6}} && (15)\\
&= 88.32 \times 10^{8}\ FLOPS/s && (16)
\end{aligned}
$$

Next, we are going to use the same GEMM kernel but change the size of the matrices. Execute the kernel but change N to 64, 128, 256, and 512.

Question 6

Load and calculate the data we need as table here.

a) Execution time (in seconds) that the kernel took
(calculated as gpu_sim_cycles/core_frequency[Hz])

b) FLOP/s as a function of problem size (N), calculated as EQUATION 14

c) IPC as a function of problem size (N), get from "gpu_ipc"

d) Simulation time as a function of problem size (N), measured by system.

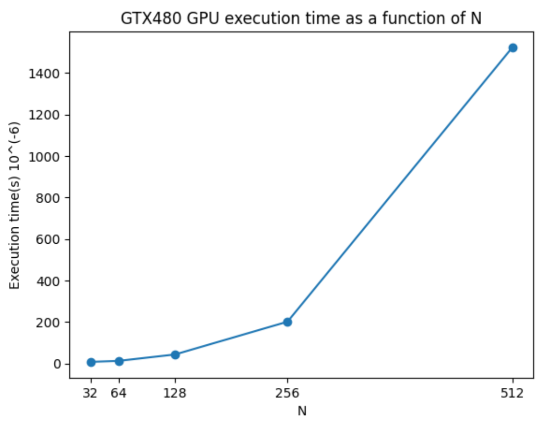
e) External bandwidth as a function of problem size (N) from the GPU characteristics, Peak off-chip DRAM bandwidth calculated by 7, L2_BW get from report

f) GPU occupancy as a function of problem size (N), which is got from report "gpu_tot_occupancy "

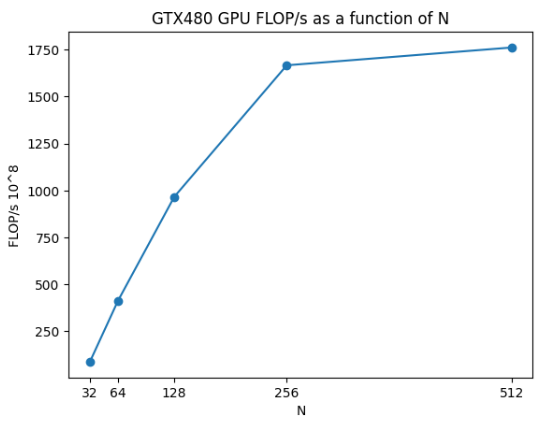
N / Performance	32	64	128	256	512
Execution time(s)$\times 10^{-6}$	7.66	12.68	43.41	201.20	1522.88
FLOP/s $\times 10^8$	88.32	413.44	966.4	1666.56	1761.28
IPC	28.80	121.61	263.82	437.72	453.30
Simulation time(s)	1.690	2.413	9.025	60.820	496.92
External bandwidth(GB/s)	177.4	177.4	177.4	177.4	177.4
Peak off-chip DRAM bandwidth(MHZ)	88704	88704	88704	88704	88704
L2_BW(GB/Sec)	1.78	6.87	14.03	22.22	22.70
GPU occupancy(%)	65.90	66.10	66.35	66.49	66.58

Table 2: Performance with N change of GTX480 GPU

Draw and discuss the following graphs:



(a) Execution time

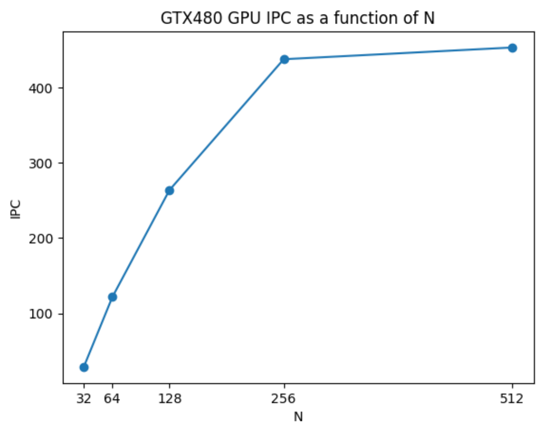


(b) FLOP/s

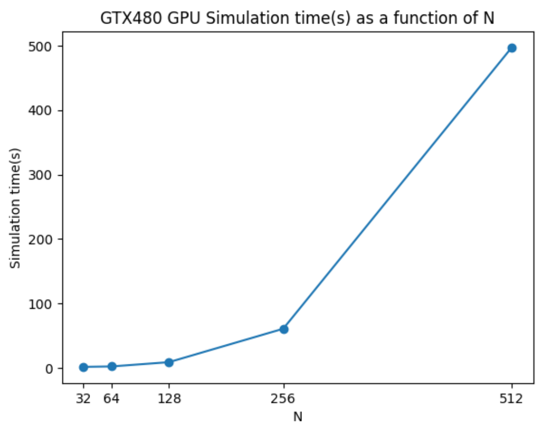
Figure 1: execution time and FLOP/s of GTX480

The Execution time rises as N increments, although the core frequency is constant, the gpu simulation cycles rises with requirement of more computation.

The FLOP/s increase as N. We can deduce from the EQUATION 13and 14, that the numerator increases with N's power of 3, greater than the execution time increase rate. But closer to the bottleneck when choosing N=512.



(a) IPC

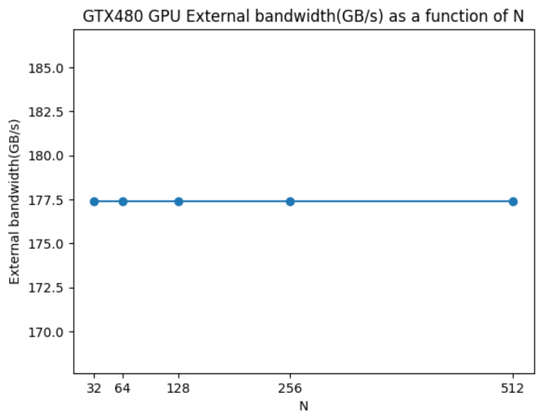


(b) simulation time

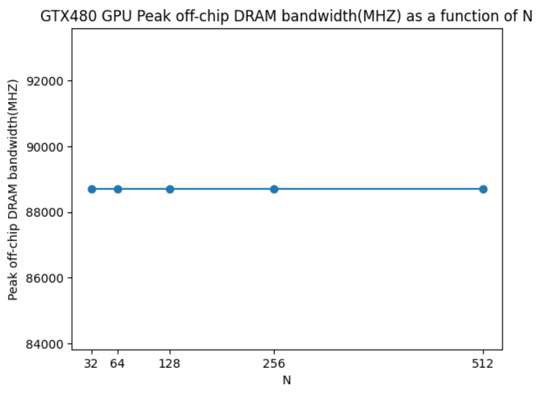
Figure 2: IPC and simulation time of GTX480

IPC is the instructions-per-cycle, which increases as N, but the growth rate is declining, may be because it is getting closer to the peak performance.

Simulation time also increases as N, and the growth rate is increasing also as N becomes bigger, dealing with larger amount of data.



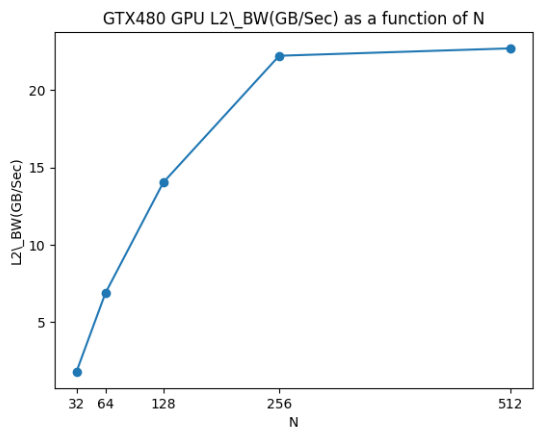
(a) External bandwidth(GB/s)



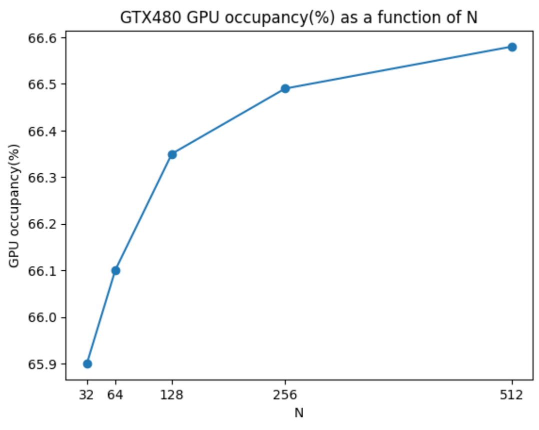
(b) Peak off-chip DRAM bandwidth(MHZ)

Figure 3: External bandwidth(GB/s) and Peak off-chip DRAM bandwidth(MHZ) of GTX480

The External bandwidth is the device Parameter of GPU GTX480, and the Peak off-chip DRAM bandwidth(MHZ) is calculated as 7, which is also a constant.



(a) L2 BW(GB/Sec)



(b) GPU occupancy(%)

Figure 4: L2 BW(GB/Sec) and GPU occupancy(%) of GTX480

L2 BW(GB/Sec) is increasing and gradually reach the bottleneck, because the amount of data increases close to the full capacity.

GPU occupancy increases as N, because the amount of computation increases as data increases, which occupy GPU more and more.

Question 7

Look up the NVIDIA RTX2060 GPU and describe its characteristics.

NVIDIA RTX2060 GPU	Release Year: January 7th, 2019
Peak performance	Pixel Rate : 80.64 GPixel/s
	Texture Rate : 201.6 GTexel/s
	FP16 (half) : 12.90 TFLOPS (2:1)
	FP32 (float) : 6.451 TFLOPS
	FP64 (double) : 201.6 GFLOPS (1:32)
External memory bandwidth	336.0 GB/s
Architecture family	Turing
Core Frequency	Base Clock : 1365 MHz
	Boost Clock : 1680 MHz
	Memory Clock: 1750 MHz 14 Gbps effective

Table 3: NVIDIA RTX2060 GPU characteristics

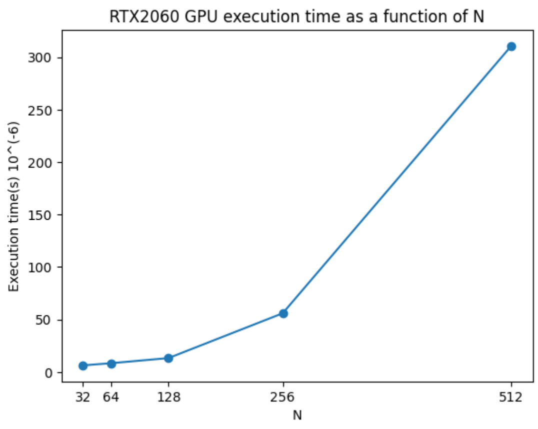
When compute a fairly small matrix multiplication of size 32x32, the simulation takes 1.223 s. The calculation process of RTX2060 is the same as the case of GTX480.

Then we load and calculate the data we need as table here.

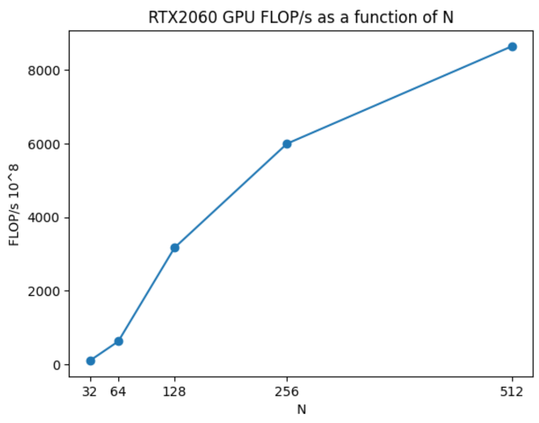
Performance \ N	32	64	128	256	512
Execution time(s)$\times 10^{-6}$	6.29	8.40	13.24	55.96	310.66
FLOP/s $\times 10^{8}$	104.32	624	3168	5995.52	8642.56
IPC	18.01	94.34	444.38	808.20	1141.19
Simulation time(s)	1.223	2.138	8.363	63.183	538.413
External bandwidth(GB/s)	336	336	336	336	336
Peak off-chip DRAM bandwidth(MHZ)	168000	168000	168000	168000	168000
L2_BW(GB/Sec)	1.95	9.76	44.56	76.12	111.39
GPU occupancy(%)	98.27	99.02	99.49	99.71	99.85

Table 4: Performance with N change of RTX2060 GPU

Draw and discuss the following graphs:



(a) Execution time

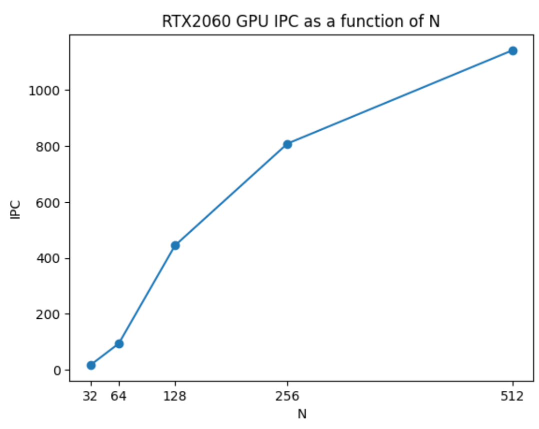


(b) FLOP/s

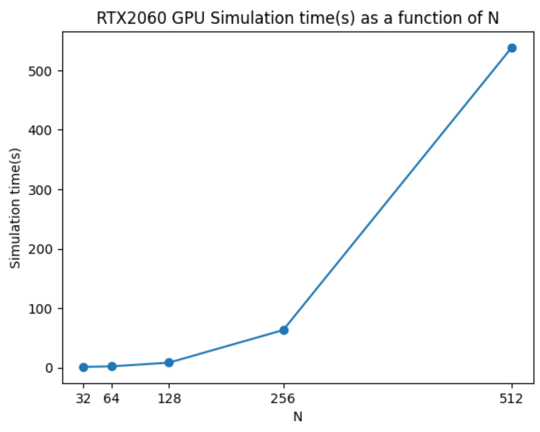
Figure 5: execution time and FLOP/s of RTX2060

The Execution time has a increase trend as N increments, just like as before. But the value of execution time of RTX2060 is less than GTX480, which brings a more efficient running speed. And the exe.Time is calculated by EQUATION 1,in which the core frequency is 1365 MHz, also higher than GTX480 with 701 MHz.

The FLOP/s increase as N. We can deduce from the EQUATION 13and 14, that the numerator increases with N's power of 3, greater than the execution time increase rate. The trend is similar to GTX480 GPU, the difference is that the FLOP/s in RTX2060 is not seen as reach to the bottleneck. It's a improvement of performance.



(a) IPC

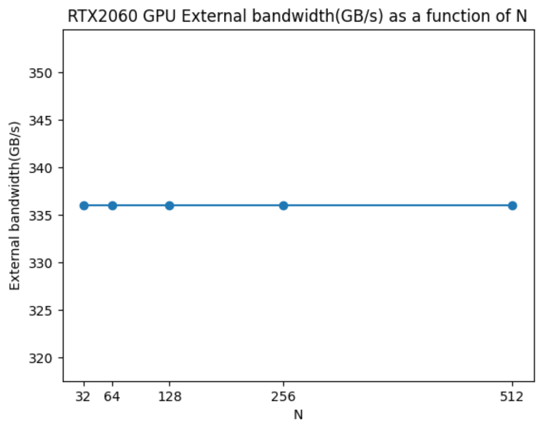


(b) simulation time

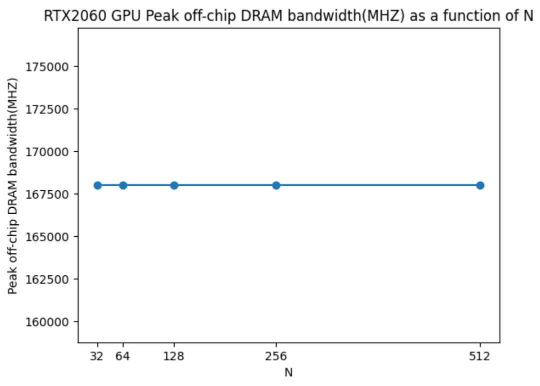
Figure 6: IPC and simulation time of RTX2060

IPC is the instructions-per-cycle, which increases as N. When N grows over 128, IPC of RTX2060 begins to be larger than GTX480, and do not reach closer to bottleneck, it may because different structure of the GPU.

Simulation time is increase as N. And the curves of both the two GPU are very similar, in terms of the curves' trend and value of points.



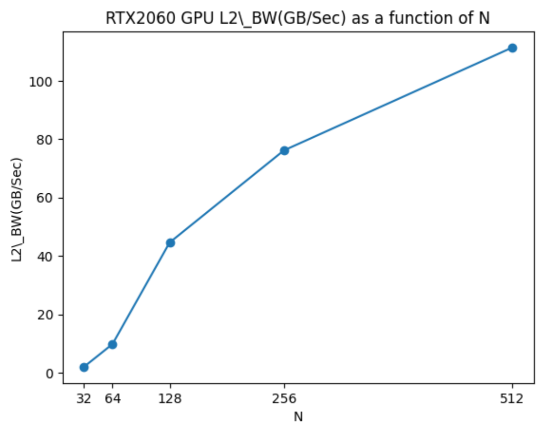
(a) External bandwidth(GB/s)



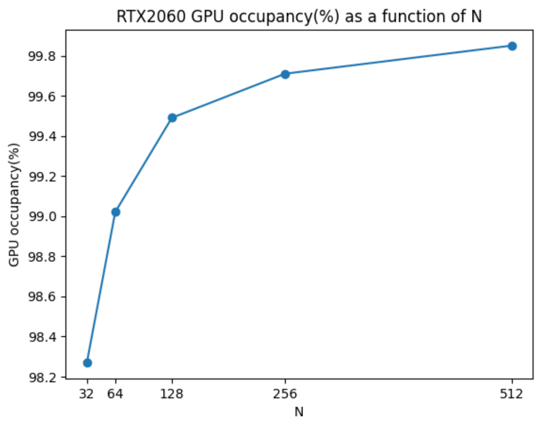
(b) Peak off-chip DRAM bandwidth(MHZ)

Figure 7: External bandwidth(GB/s) and Peak off-chip DRAM bandwidth(MHZ) of RTX2060

The External bandwidth is the device Parameter of GPU RTX2060, and the Peak off-chip DRAM bandwidth(MHZ) is calculated as 7, which is also a constant.



(a) L2 BW(GB/Sec)



(b) GPU occupancy(%)

Figure 8: L2 BW(GB/Sec) and GPU occupancy(%) of RTX2060

L2 BW(GB/Sec) is increasing. The difference with GTX480 is that, RTX2060's L2 BW is not reach to the bottleneck, remaining a trend of growth, that expresses a improvement of the capacity and performance of RTX2060 GPU.

GPU occupancy increases as N, and the values are very close to 100%, the occupancy of resource is more optimal than GTX480 GPU, which has occupancy only around 65%.

Exercise 3: Power Simulation (Directed part)

Question 8

Now use the GTX480 GPU model to execute and investigate the power consumption of three different applications.

a) Min, max, and average power consumption for the three applications. Since the applications execute multiple kernels several times, we compute the average value of all min power consumptions to be the final value that we use, and list in the table below. Similarly, we compute the average value of max and avg. power consumption.

b) Execution time (in seconds) that the three applications took (ignoring host overhead), Some applications might execute multiple kernels several times. Use the final *gpu_tot_sim_cycle* (instead of *gpu_sim_cycle*) metric for those.

$$Execution\ time = \frac{gpu\_tot\_sim\_cycle}{frequency} \tag{17}$$

c) Energy consumption (Joules) of the three applications. Joules is the same as Watt-seconds (W * s). It is calculated as

$$W \times s = average\ power\ consumption \times Execution\ time \tag{18}$$

	maxPower	avgPower	minPower	exeTime(s)$\times 10^{-6}$	EnergyConspt.(J)
HotSpot3D	152.26	110.94	15.25	3640	0.4038
BFS	109.14	53.52	15.25	6849	0.3666
LUD	157.39	46.13	12.43	169467	7.817

Table 5: power consumption of three different applications

Draw and discuss the following graphs:

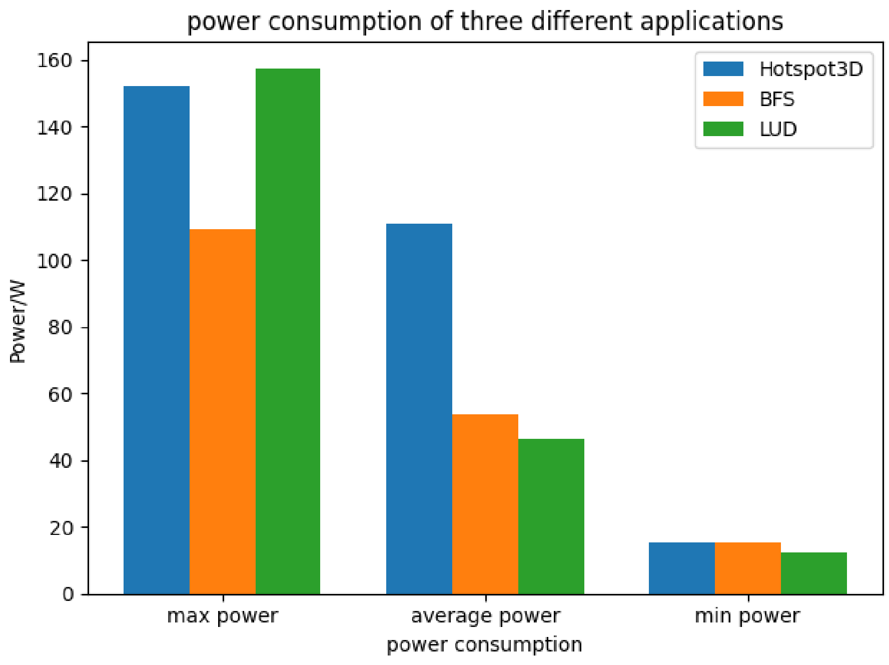
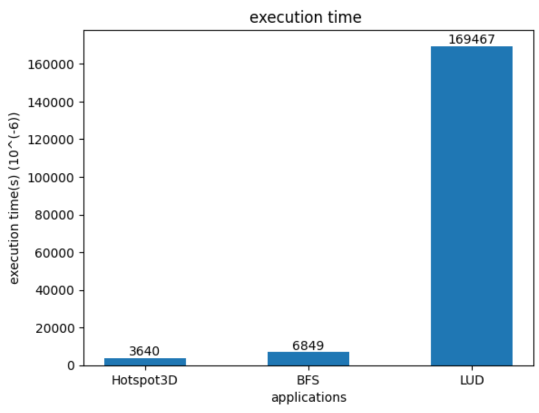
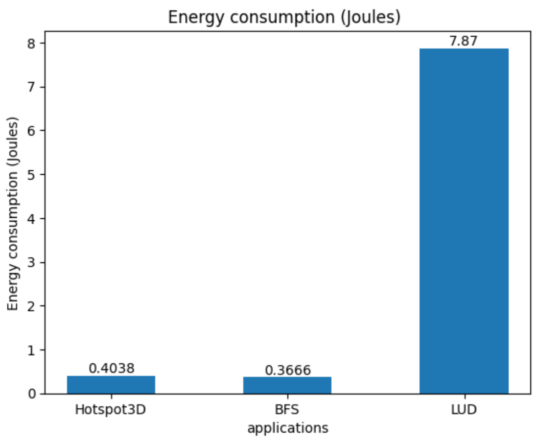


Figure 9: power consumption

We can see that the LUD power consumption has the biggest fluctuation among the three applications. And BFS has the most stable power variation.



(a) Execution time(s)$\times 10^{-6}$



(b) energy consumption(J)

Figure 10: Execution time and energy consumption

LUD has the largest execution time of $169467 \times 10^{-6} seconds$, approximately 0.17 s, while Hotspot3D has the smallest.

When it comes to energy consumption, which is calculated by EQUATION 18. This value is affected by both execution time and power consumption. So, although LUD has a smallest average power consumption of 46.13 W, with its highest execution time, it can still get the higest energy consumption of 7.87 J.

Question 9

Use the GTX480 model and change the core frequency-, interconnect-,L2 clocks to 300 MHz and 1GHz respectively. Keep the DRAM clock unmodified. Redo question 8 but with the new clock frequencies.

change the core frequency-, interconnect-,L2 clocks to 300 MHz

	maxPower	avgPower	minPower	exeTime(s)$\times 10^{-6}$	EnergyConspt.(J)
HotSpot3D	71.10	56.08	12.88	8095	0.4540
BFS	72.39	31.54	12.88	16018	0.5052
LUD	79.26	31.12	11.91	391452	12.1820

Table 6: power consumption choosing 300MHz

Draw and discuss the following graphs:

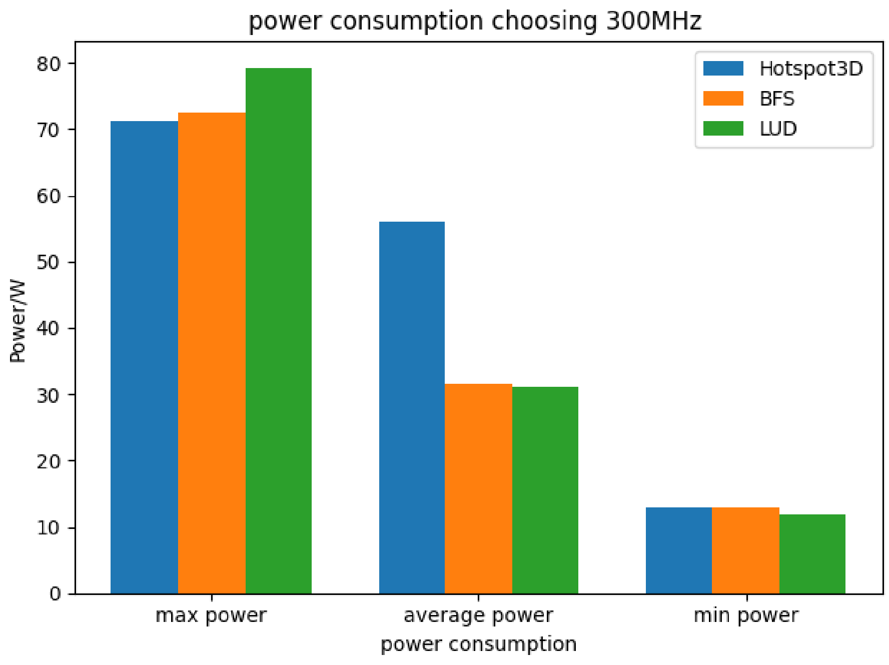
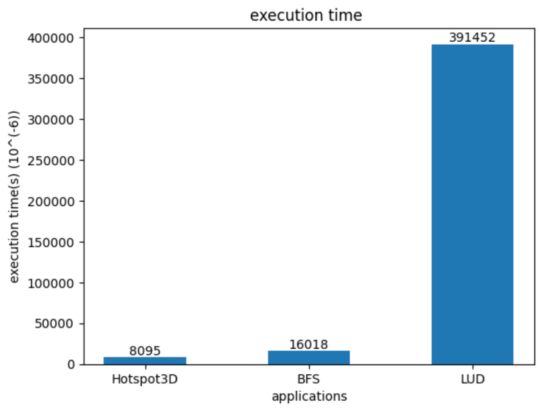
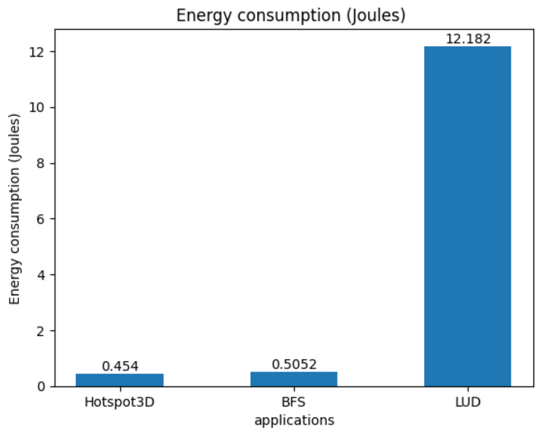


Figure 11: power consumption

We can see that the LUD power consumption has the biggest fluctuation among the three applications. And BFS has the most stable power variation.



(a) Execution time(s)$\times 10^{-6}$



(b) energy consumption(J)

Figure 12: Execution time and energy consumption 300MHz

LUD has the largest execution time approximately 0.39 s, while Hotspot3D has the smallest.

When it comes to energy consumption, LUD gets the higest energy consumption of 12.182 J.

change the core frequency-, interconnect-,L2 clocks to 1GHz

	maxPower	avgPower	minPower	exeTime(s)$\times 10^{-6}$	EnergyConspt.(J)
HotSpot3D	203.80	137.42	17.05	2870	0.3944
BFS	144.40	70.03	17.05	4858	0.3402
LUD	205.47	57.08	13.41	119966	6.848

Table 7: power consumption choosing 1GHz

Draw and discuss the following graphs:

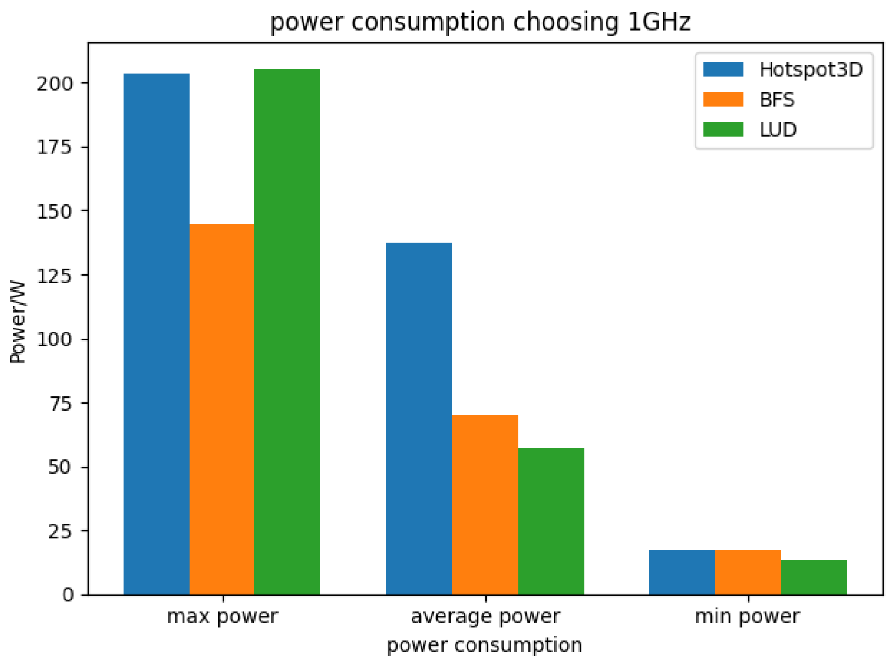
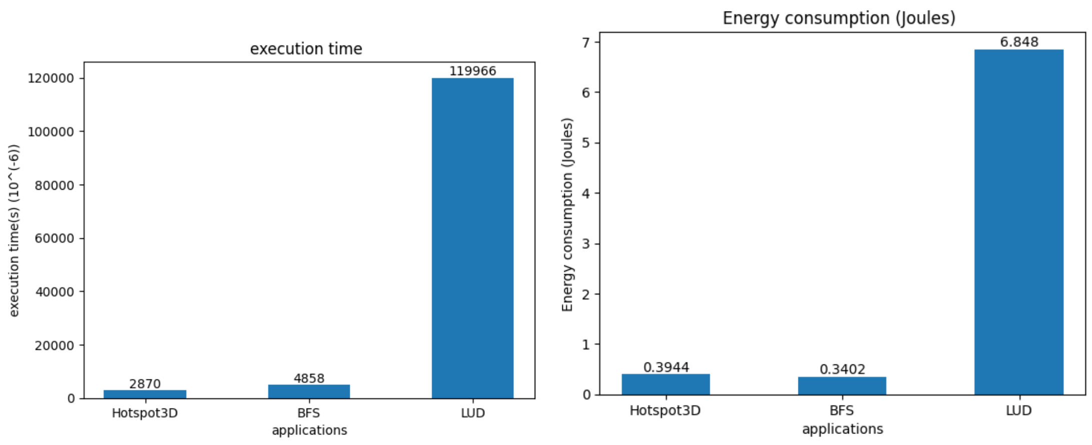


Figure 13: power consumption

We can see that the LUD power consumption has the biggest fluctuation among the three applications. And BFS has the most stable power variation.



(a) Execution time(s)$\times 10^{-6}$ (b) energy consumption(J)

Figure 14: Execution time and energy consumption 1GHz

LUD has the largest execution time approximately 0.12 s, while Hotspot3D has the smallest.

When it comes to energy consumption, LUD getS the higest energy consumption of 6.848 J.

Summarize above simulations with case 300MHz, 700MHz and 1GHz, we get that when the frequency increases, the execution time decreases, power increases, and energy consumption decrease.

To determine if the applications is **compute- or memory-bound**, we think that: if frequency change from 300MHz to 1GHz(3.33 times speeder), the execution time should speedup about 3.33 times, then this application should be compute-bound. Otherwise, if the execution time does not change much as the frequency change, then we say this application should be memory-bound.

Based on this thought, comparing FIGURE 12a and 14a, we determine that all of the three applications are compute-bound.

Exercise 4: Examining an Application of your choice! (Open Ended Part)

Testing application selection

We compiled the linear equation solver by gaussian elimination from Rodinia.

- Benchmark suite: Rodinia
- Application: Gaussian elimination
- Input size: 128
- Time complexity: O(n^3)

Algorithm analysis

Gaussian elimination has two main parts that can be done in parallel:
1. Calculate the multiplier (parallel in row)
2. Calculate the upper triangular matrix (parallel in row and column)

Which corresponds to the two kernels of the original cuda code:
1. __global__ void Fan1(float *m, float *a, int Size, int t);

```
__global__ void Fan1(float *m_cuda, float *a_cuda, int Size, int
   t)
{
        if(threadIdx.x + blockIdx.x * blockDim.x >= Size-1-t)
           return;
        *(m_cuda+Size*(blockDim.x*blockIdx.x+threadIdx.x+t+1)+t)
           = *(a_cuda+Size*(blockDim.x*blockIdx.x+threadIdx.x+t
           +1)+t) / *(a_cuda+Size*t+t);
}
```

2. __global__ void Fan2(float *m_cuda, float *a_cuda, float *b_cuda,int Size, int j1, int t)

```
__global__ void Fan2(float *m_cuda, float *a_cuda, float *b_cuda,
   int Size, int j1, int t)
{
        if(threadIdx.x + blockIdx.x * blockDim.x >= Size-1-t)
           return;
        if(threadIdx.y + blockIdx.y * blockDim.y >= Size-t)
           return;

        int xidx = blockIdx.x * blockDim.x + threadIdx.x;
        int yidx = blockIdx.y * blockDim.y + threadIdx.y;
        a_cuda[Size*(xidx+1+t)+(yidx+t)] -= m_cuda[Size*(xidx+1+t
           )+t] * a_cuda[Size*t+(yidx+t)];
        if(yidx == 0){
                b_cuda[xidx+1+t] -= m_cuda[Size*(xidx+1+t)+(yidx+
                   t)] * b_cuda[t];
        }
}
```

And the 2 kernels are called and synchronized N times(for a N by N linear equation) on a host:

```
for (t=0; t<(Size-1); t++) {
    Fan1<<<dimGrid,dimBlock>>>(m_cuda,a_cuda,Size,t);
    cudaThreadSynchronize();
    Fan2<<<dimGridXY,dimBlockXY>>>(m_cuda,a_cuda,b_cuda,Size,Size
       -t,t);
    cudaThreadSynchronize();
    checkCUDAError("Fan2");
}
```

For each iteration, Fan1 will execute $N-1-t$ FP operations, Fan2 will execute $2*(N-1-t)*(N-t+1)$ FP operations. Sum over t from 0 to $N-2$, we can get the total FP operations as follows:

$$Total\ FPs = \frac{(N-1)(4N^2+13N)}{6} = O(N^3) \tag{19}$$

Comparison between models

We choose GTX480, RTX2060, TitanV, QV100 as the models to be tested, their parameters listed below:

Model	Arch	Freq(MHz)	BW(GB/s)	Cores	Released
GTX480	Fermi	700	177.4	15*1	2010
RTX2060	Turing	1365	366	30*1	2019
TitanV	Volta	1200	651.3	40*2	2017
QV100	Volta	1132	868.4	80*1	2018

Table 8: Parameters for different GPU models

We can see from the table, as a card that has been released more than 10 years, GTX480's spec is way behind that of the newer version. But it can be used as the baseline.

TitanV and QV100 as two high-end graphic card, they have the top-level hardware specifications. Their structure is slightly different: TitanV has 2 clusters of cores while QV100 has only 1 clusters with 80 cores. TitanV has higher frequency but QV100 has larger bandwidth.

Compared to these two, RTX2060 only has less than half cores. But it is still leading in its higher frequency.

Testing result

Because each kernel needs to be called $N-1$ times, the total simulations in gpgpusim are $2N-2$ times. So we collected all the data, sum up the gpu_tot_sim_cycle to get the total execution time, average others to get the average performance for other metrics(FLOPs = 1422400):

Model	Sim Cycles	IPC	Occupancy	L2 BW	Time(s)	MFLOP/s
gtx480	99461585	76.50	16.15%	62.98	0.1420	10.02
qv100	199781642	42.74	12.43%	23.90	0.1765	8.06
rtx2060	202255868	42.15	62.93%	30.20	0.1482	9.60
titanV	39858355	205.41	12.77%	120.58	0.0332	42.84

Table 9: Performance Metrics for Different GPU Models

As the flagship of NVIDIA of 2017, TitanV is way more faster than other three models. Its IPC reaches a surprisingly 5 times higher than its following model RTX2060 and qv100. We think it benefits from the dual-cluster design and comparably high frequency.

Surprisingly, although GTX480 has a relatively slower hardware to other models, it still gets the second place. But the other three models' difference is not that significant as with the titanV. So it's possibly because the testing application hasn't reached the bottleneck even for the oldest version.

In conclusion, if the scale of the linear equation is not that huge, it's a good idea to choose GTX480 in terms of economical consideration. But if you want to pursue the extreme performance and fastest calculation time, titanV will be a good choice.

Exercise 5: Architectural Exploration (Open Ended Part)

Baseline

We use the gtx480 model as the baseline model. Its core frequency is 700MHz, number of SM cores is 16, number of warps per SM is 32. The maximum possible warps per SM is 48(probably constrained by the shader structure of GTX480).

Power consumption analysis

We refered to the GPU wattch manual. It emulates NVIDIA GPU based on the following components:

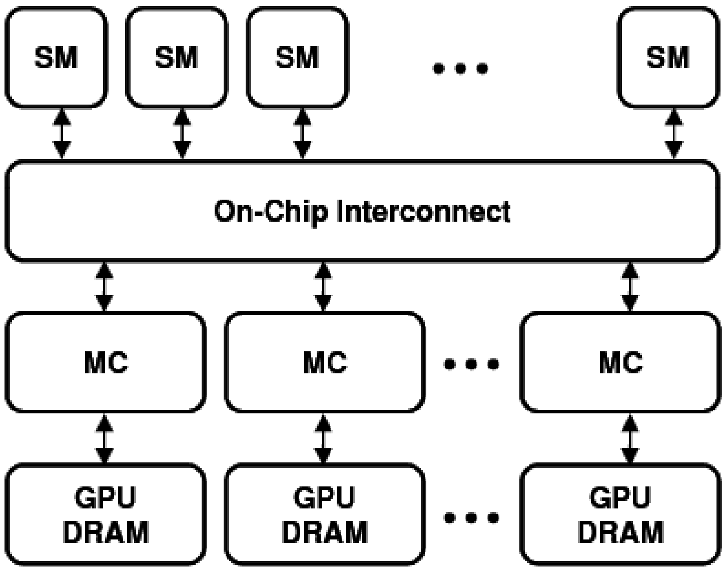
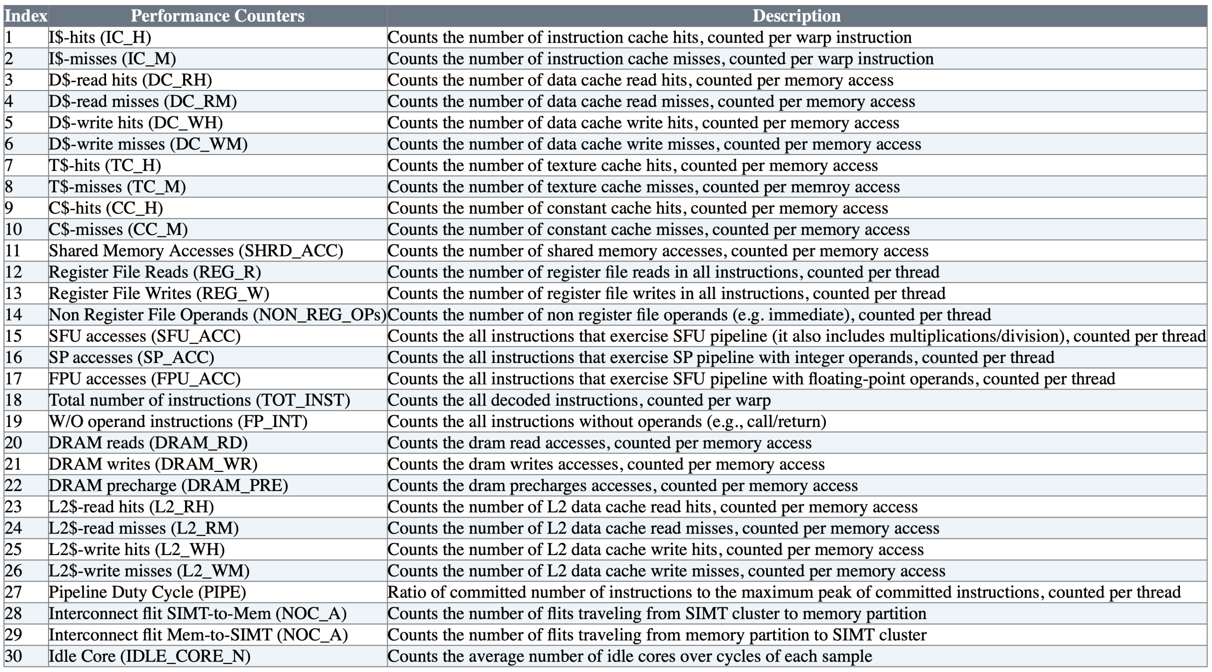


Figure 15: Main components modeled in the GPUWattch

And it calcualtes the total power according to sets of performance counters:



Index	Performance Counters	Description
1	I$-hits (IC_H)	Counts the number of instruction cache hits, counted per warp instruction
2	I$-misses (IC_M)	Counts the number of instruction cache misses, counted per warp instruction
3	D$-read hits (DC_RH)	Counts the number of data cache read hits, counted per memory access
4	D$-read misses (DC_RM)	Counts the number of data cache read misses, counted per memory access
5	D$-write hits (DC_WH)	Counts the number of data cache write hits, counted per memory access
6	D$-write misses (DC_WM)	Counts the number of data cache write misses, counted per memory access
7	T$-hits (TC_H)	Counts the number of texture cache hits, counted per memory access
8	T$-misses (TC_M)	Counts the number of texture cache misses, counted per memroy access
9	C$-hits (CC_H)	Counts the number of constant cache hits, counted per memory access
10	C$-misses (CC_M)	Counts the number of constant cache misses, counted per memory access
11	Shared Memory Accesses (SHRD_ACC)	Counts the number of shared memory accesses, counted per memory access
12	Register File Reads (REG_R)	Counts the number of register file reads in all instructions, counted per thread
13	Register File Writes (REG_W)	Counts the number of register file writes in all instructions, counted per thread
14	Non Register File Operands (NON_REG_OPs)	Counts the number of non register file operands (e.g. immediate), counted per thread
15	SFU accesses (SFU_ACC)	Counts the all instructions that exercise SFU pipeline (it also includes multiplications/division), counted per thread
16	SP accesses (SP_ACC)	Counts the all instructions that exercise SP pipeline with integer operands, counted per thread
17	FPU accesses (FPU_ACC)	Counts the all instructions that exercise SFU pipeline with floating-point operands, counted per thread
18	Total number of instructions (TOT_INST)	Counts the all decoded instructions, counted per warp
19	W/O operand instructions (FP_INT)	Counts the all instructions without operands (e.g., call/return)
20	DRAM reads (DRAM_RD)	Counts the dram read accesses, counted per memory access
21	DRAM writes (DRAM_WR)	Counts the dram writes accesses, counted per memory access
22	DRAM precharge (DRAM_PRE)	Counts the dram precharges accesses, counted per memory access
23	L2$-read hits (L2_RH)	Counts the number of L2 data cache read hits, counted per memory access
24	L2$-read misses (L2_RM)	Counts the number of L2 data cache read misses, counted per memory access
25	L2$-write hits (L2_WH)	Counts the number of L2 data cache write hits, counted per memory access
26	L2$-write misses (L2_WM)	Counts the number of L2 data cache write misses, counted per memory access
27	Pipeline Duty Cycle (PIPE)	Ratio of committed number of instructions to the maximum peak of committed instructions, counted per thread
28	Interconnect flit SIMT-to-Mem (NOC_A)	Counts the number of flits traveling from SIMT cluster to memory partition
29	Interconnect flit Mem-to-SIMT (NOC_A)	Counts the number of flits traveling from memory partition to SIMT cluster
30	Idle Core (IDLE_CORE_N)	Counts the average number of idle cores over cycles of each sample

Figure 16: Performance counter in the GPUWattch

From the performance counter we could see that the main factors that affect the total power consumption include cache hit rate, total arithmetic operations been taken and the number of idle cores.

As we have learned, the frequency will affect the dynamic power consumption for $P_{dynamic} = Cf^2V$. And the higher frequency can reduce the total execution time. The same goes with the number of cores and the number of warps per core. But more cores and more warps per core may increase the hardware complexity so to increase the static power consumption. We can only discuss their relation in our specific application.

Testing result

We tested the gaussian elimination(same as previous section) on different architecture settings. We averaged the power statistics of each calling to the kernels and sum up its cycles to get the total execution time.

Firstly, we tested the influence of the frequency, where we use 3 sets of frequencies from 300MHz to 1000MHz. The average power for 1000MHz is more than twice of 300MHz, but the total execution time is about 0.3 times of it. As a result, the core frequency of 1000MHz has the lowest energy consumption:

	300	**700(Default)**	**1000**
tot_avg	28.41	46.19	59.41
tot_min	17.24	26.2	29.34
tot_max	37.95	70.27	95.81
cycles	99317666	99461585	99717986
execution time	0.3311	0.1421	0.0997
joules	9.41	6.56	5.92

Table 10: Energy table w.r.t Frequency

Then we tested the power relation between number of cores and number of warps per core. It can be seen that the total cycles remains the same, means the same execution time. With less hardware complexity, the total energy consumption will be less correspondingly:

	4	**8**	**32**
tot_avg	42.95	44.03	50.51
tot_min	25.93	26.02	26.56
tot_max	61.05	64.06	83.02
cycles	99461585	99461585	99461585
execution time	0.142	0.142	0.142
joules	6.1	6.26	7.18

Table 11: Energy table w.r.t Cores

	8	**16**	**48**
tot_avg	43.36	44.31	48.06
tot_min	25.96	26.04	26.35
tot_max	62.16	64.85	75.75
cycles	99461585	99461585	99461585
execution time	0.142	0.142	0.142
joules	6.16	6.3	6.83

Table 12: Energy table w.r.t Warp

As energy-efficient as possible

Because the limitation of time, we can't test all possible settings. But for all the parameters we have tested, we evaluate the cases and get a possible "most" energy-efficient configuration:

Freq	**#Cores**	**Warp**
1000MHz	4	8

Table 13: Result with the possible best parameter

tot_avg	**tot_min**	**tot_max**	**cycles**	**execution**	**joules**
53.69	28.74	79.45	99717986.0	0.0997	5.35

Table 14: Result with the possible best parameter

When we choose the "best" configurations, the total joules got further decreased. It's possibly because the computation and memory bounds are not reached for the input size—the execution time decrease linearly when the frequency increase and less hardware resource been used can reduce the static power consumption.

In real world, we need consider more testing cases and more sophisticated configurations to get a more accurate evaluation.