

# Performance Considerations

## 1 Dynamic Partitioning of Resources

- streaming multiprocessor resources
- the CUDA occupancy calculator

## 2 the Compute Visual Profiler

- getting started with `computeprof`
- analysis of the kernel `matrixMul`

## 3 Data Prefetching and Instruction Mix

- registers between global and shared memory
- maximizing instruction throughput

MCS 572 Lecture 36  
Introduction to Supercomputing  
Jan Verschelde, 14 November 2016

# Performance Considerations

## 1 Dynamic Partitioning of Resources

- streaming multiprocessor resources
- the CUDA occupancy calculator

## 2 the Compute Visual Profiler

- getting started with `computeprof`
- analysis of the kernel `matrixMul`

## 3 Data Prefetching and Instruction Mix

- registers between global and shared memory
- maximizing instruction throughput

## streaming multiprocessor resources – part I

Comparing GPUs with respective compute capabilities 1.1, 2.0, 3.5, and 6.0: GeForce 9400M, Tesla C2050/C2070, K20C, P100:

compute capability	1.1	2.0	3.5	6.0
maximum number of threads per block	512	1,024		
maximum number of resident blocks per streaming multiprocessor	8	16	32	
warp size		32		
maximum number of resident warps per streaming multiprocessor	24	48	64	
maximum number of resident threads per streaming multiprocessor	768	1,536	2,048	

data in the table from the CUDA C Programming Guide appendix G

## dynamic partitioning of thread slots

During runtime, thread slots are partitioned and assigned to thread blocks.

Streaming multiprocessors are versatile by their ability to dynamically partition the thread slots among thread blocks.

They can

- either execute many thread blocks of few threads each,
- or execute a few thread blocks of many threads each.

In contrast, fixed partitioning where the number of blocks and threads per block are fixed will lead to waste.

Goal: keep multiprocessors fully occupied.

## interactions between resource limitations – C2050

The Tesla C2050/C2070 has 1,536 thread slots per streaming multiprocessor. As  $1,536 = 32 \times 48$ , we have

$$\text{number of thread slots} = \text{warp size} \times \text{number of warps per block.}$$

For 32 threads per block, we have  $1,536/32 = 48$  blocks

$\leftrightarrow$  at most 8 blocks per streaming multiprocessor.

To fully utilize both the block and thread slots, to have 8 blocks, we should have

- $1,536/8 = 192$  threads per block, or
- $192/32 = 6$  warps per block.

## interactions between resource limitations – K20C

The K20C has 2,048 thread slots per streaming multiprocessor.  
As  $2,048 = 32 \times 64$ , we have

$$\text{number of thread slots} = \text{warp size} \times \text{number of warps per block.}$$

For 32 threads per block, we have  $2,048/32 = 64$  blocks  
 $\leftrightarrow$  at most 16 blocks per streaming multiprocessor.

To fully utilize both the block and thread slots,  
to have 16 blocks, we should have

- $2,048/16 = 128$  threads per block, or
- $128/32 = 4$  warps per block.

## interactions between resource limitations – P100

The P100 has 2,048 thread slots per streaming multiprocessor.  
As  $2,048 = 32 \times 64$ , we have

$$\text{number of thread slots} = \text{warp size} \times \text{number of warps per block.}$$

For 32 threads per block, we have  $2,048/32 = 64$  blocks  
 $\leftrightarrow$  at most 32 blocks per streaming multiprocessor.

To fully utilize both the block and thread slots,  
to have 32 blocks, we should have

- $2,048/32 = 64$  threads per block, or
- $64/32 = 2$  warps per block.

## streaming multiprocessor resources – part II

Comparing GPUs with respective compute capabilities 1.1, 2.0, 3.5, and 6.0: GeForce 9400M, Tesla C2050/C2070, K20C, and P100.

compute capability	1.1	2.0	3.5	6.0
number of 32-bit registers per streaming multiprocessor	8K	32K	64K	
maximum amount of shared memory per streaming multiprocessor	16KB	48KB	64KB	
number of shared memory banks	16	32		
amount of local memory per thread	16KB	512KB		
constant memory size		64KB		
cache working set for constant memory per streaming memory		8KB		10KB

Local memory resides in device memory, so local memory accesses have the same high latency and low bandwidth as global memory.

## dynamic partitioning of resources

Registers hold frequently used programmer and compiler-generated variables to reduce access latency and conserve memory bandwidth.

Variables in a kernel that are not arrays are automatically placed into registers.

By dynamically partitioning the registers among blocks, a streaming multiprocessor can accommodate

- more blocks if they require few registers, and
- fewer blocks if they require many registers.

As with block and thread slots, there is a potential interaction between register limitations and other resource limitations.

## interactions between resource limitations

Consider the matrix-matrix multiplication example. Assume

- the kernel uses 21 registers, and
- we have 16-by-16 thread blocks.

How many threads can run on each Streaming Multiprocessor (SM)?

- ① We calculate the number of registers for each block:

$$16 \times 16 \times 21 = 5,376 \text{ registers.}$$

- ② We have  $32 \times 1,024$  registers per SM:

$$32 \times 1,024 / 5,376 = 6 \text{ blocks}$$

and  $6 < 8$  = the maximum number of blocks per SM.

- ③ We calculate the number of threads per SM:

$$16 \times 16 \times 6 = 1,536 \text{ threads}$$

and we can have at most 1,536 threads per SM.

## a performance cliff

Suppose we use one extra register, 22 instead of 21.

- ① We calculate the number of registers for each block:

$$16 \times 16 \times 22 = 5,632 \text{ registers.}$$

- ② We have  $32 \times 1,024$  registers per SM:

$$32 \times 1,024 / 5,632 = 5 \text{ blocks.}$$

- ③ We calculate the number of threads per SM:

$$16 \times 16 \times 5 = 1,280 \text{ threads}$$

and with 21 registers we could use all 1,536 threads per SM.

Adding one register led to a reduction of 17% in the parallelism.

When a slight increase in one resource leads to a dramatic reduction in parallelism and performance, one speaks of a *performance cliff*.

# Performance Considerations

## 1 Dynamic Partitioning of Resources

- streaming multiprocessor resources
- the CUDA occupancy calculator

## 2 the Compute Visual Profiler

- getting started with `computeprof`
- analysis of the kernel `matrixMul`

## 3 Data Prefetching and Instruction Mix

- registers between global and shared memory
- maximizing instruction throughput

# spreadsheet in /usr/local/cuda/tools

## CUDA GPU Occupancy Calculator

Just follow steps 1, 2, and 3 below! (or click here for help)

1.) Select Compute Capability (click): 3.5

1.b) Select Shared Memory Size Config (bytes): 49152

2.) Enter your resource usage:

Threads Per Block: 256

Registers Per Thread: 32

Shared Memory Per Block (bytes): 4096

(Don't edit anything below this line)

3.) GPU Occupancy Data is displayed here and in the graphs:

Active Threads per Multiprocessor: 2048

Active Warps per Multiprocessor: 64

Active Thread Blocks per Multiprocessor: 8

Occupancy of each Multiprocessor: 100%

Physical Limits for GPU Compute Capability:

3.5

Threads per Warp: 32

Warp per Multiprocessor: 64

Threads per Multiprocessor: 2048

Thread Blocks per Multiprocessor: 16

Total # of 32-bit registers per Multiprocessor: 65536

Register allocation unit size: 256

Register allocation granularity: warp

Registers per Thread: 256

Shared Memory per Multiprocessor (bytes): 49152

Shared Memory Allocation unit size: 256

Warp allocation granularity: 4

Maximum Thread Block Size: 1024

Allocated Resources

Per Block Limit Per SM = Allocatable Blocks Per SM

Warp (Threads Per Block / Threads Per Warp): 6 64 6

Registers (Warp limit per SM due to per-warp reg count): 6 64 6

Shared Memory (Bytes): 4096 49152 12

Note: SM is an abbreviation for (Single) Multiprocessor

Maximum Thread Blocks Per Multiprocessor

Blocks/SM \* Warps/Block = Warps/SM

Limited by Max Warps or Max Blocks per Multiprocessor: 8 8 64

Limited by Registers per Multiprocessor: 8 8 64

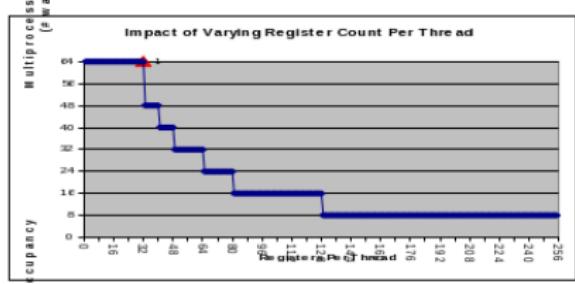
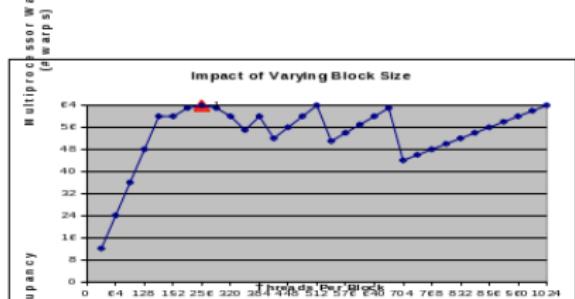
Limited by Shared Memory per Multiprocessor: 12 12 12

Note: Occupancy limit is shown in orange

Physical Max Warps/SM = 64  
Occupancy = 64 / 64 = 100%

[Click Here for detailed instructions on how to use this occupancy calculator](#)  
[For more information on NVIDIA CUDA, visit http://developer.nvidia.com/cuda](http://developer.nvidia.com/cuda)

Your chosen resource usage is indicated by the red triangle on the graphs. The other data points represent the range of possible block sizes, register counts, and shared memory allocation.



# Performance Considerations

## 1 Dynamic Partitioning of Resources

- streaming multiprocessor resources
- the CUDA occupancy calculator

## 2 the Compute Visual Profiler

- getting started with `computeprof`
- analysis of the kernel `matrixMul`

## 3 Data Prefetching and Instruction Mix

- registers between global and shared memory
- maximizing instruction throughput

## getting started with computeprof

Compute Visual Profiler is a graphical user interface based profiling tool to measure performance and to find potential opportunities for optimization in order to achieve maximum performance.

Login to kepler with ssh -X and go the directory  
/usr/local/cuda/bin/computeprof  
to launch the program computeprof.

We look at one of the example projects matrixMul.

# Performance Considerations

## 1 Dynamic Partitioning of Resources

- streaming multiprocessor resources
- the CUDA occupancy calculator

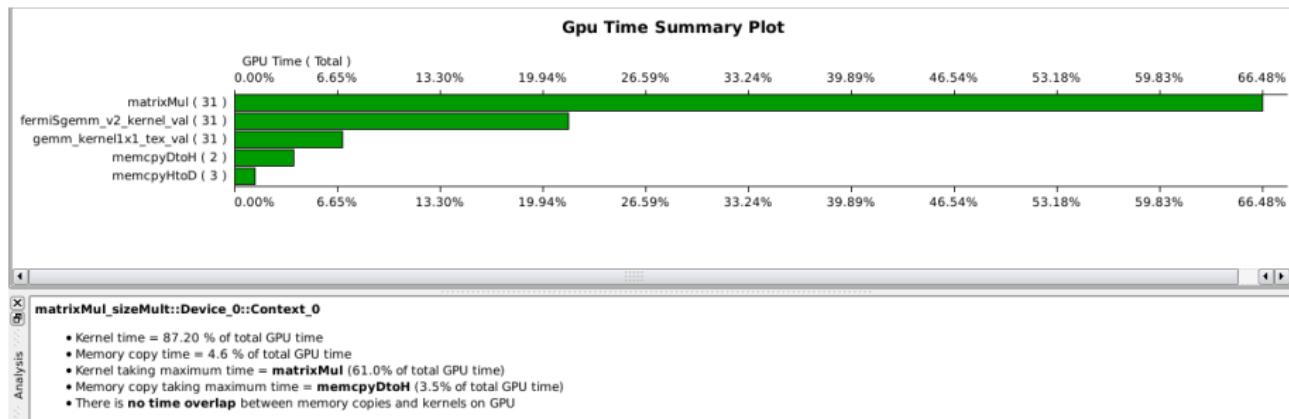
## 2 the Compute Visual Profiler

- getting started with `computeprof`
- analysis of the kernel `matrixMul`

## 3 Data Prefetching and Instruction Mix

- registers between global and shared memory
- maximizing instruction throughput

# GPU time summary



# limiting factor identification

## Analysis for kernel matrixMul on device Tesla C2050

### Summary profiling information for the kernel:

Number of calls: 31

Minimum GPU time(us): 4184.67

Maximum GPU time(us): 4192.67

Average GPU time(us): 4188.50

GPU time (%): 61.04

Grid size: [20 30 1]

Block size: [32 32 1]

### Limiting Factor

Achieved Instruction Per Byte Ratio: 10.87 ( Balanced Instruction Per Byte Ratio: 3.57 )

Achieved Occupancy: 0.67 ( Theoretical Occupancy: 0.67 )

IPC: 1.02 ( Maximum IPC: 2 )

Achieved global memory throughput: 10.00 ( Peak global memory throughput(GB/s): 144.00 )

IPC = Instructions Per Cycle

# memory throughput analysis

## Memory Throughput Analysis for kernel matrixMul on device Tesla C2050

- Kernel requested global memory read throughput(GB/s): 23.47
- Kernel requested global memory write throughput(GB/s): 0.59
- Kernel requested global memory throughput(GB/s): 24.06
- L1 cache read throughput(GB/s): 23.47
- L1 cache global hit ratio (%): 0.00
- Texture cache memory throughput(GB/s): 0.00
- Texture cache hit rate(%): 0.00
- L2 cache texture memory read throughput(GB/s): 0.00
- L2 cache global memory read throughput(GB/s): 23.47
- L2 cache global memory write throughput(GB/s): 0.59
- L2 cache global memory throughput(GB/s): 24.06
- Local memory bus traffic(%): 0.00
- Global memory excess load(%): 0.00
- Global memory excess store(%): 0.00
- Achieved global memory read throughput(GB/s): 9.27
- Achieved global memory write throughput(GB/s): 0.73
- Achieved global memory throughput(GB/s): 10.00
- Peak global memory throughput(GB/s): 144.00

# instruction throughput analysis

---

## Instruction Throughput Analysis for kernel matrixMul on device Tesla C2050

- IPC: 1.02
- Maximum IPC: 2
- Divergent branches(%): 0.00
- Control flow divergence(%): 0.04
- Replayed Instructions(%): 0.57
  - Global memory replay(%): 2.25
  - Local memory replays(%): 0.00
  - Shared bank conflict replay(%): 0.00
- Shared memory bank conflict per shared memory instruction(%): 0.00

IPC = Instructions Per Cycle

# occupancy analysis

---

## Occupancy Analysis for kernel matrixMul on device Tesla C2050

- Kernel details: Grid size: [20 30 1], Block size: [32 32 1]
- Register Ratio: 0.8125 ( 26624 / 32768 ) [25 registers per thread]
- Shared Memory Ratio: 0.166667 ( 8192 / 49152 ) [8192 bytes per Block]
- Active Blocks per SM: 1 (Maximum Active Blocks per SM: 8)
- Active threads per SM: 1024 (Maximum Active threads per SM: 1536)
- Potential Occupancy: 0.666667 ( 32 / 48 )
- Occupancy limiting factor: Block-Size

# Performance Considerations

## 1 Dynamic Partitioning of Resources

- streaming multiprocessor resources
- the CUDA occupancy calculator

## 2 the Compute Visual Profiler

- getting started with `computeprof`
- analysis of the kernel `matrixMul`

## 3 Data Prefetching and Instruction Mix

- registers between global and shared memory
- maximizing instruction throughput

## accessing global memory

One of the most important resource limitations is access to global memory and long latencies.

Scheduling other warps while waiting for memory access is powerful, but often not enough.

A complementary to warp scheduling solution is to prefetch the next data elements while processing the current data elements.

Combined with tiling, data prefetching provides extra independent instructions to enable the scheduling of more warps to tolerate long memory access latencies.

## prefetching in registers

For the tiled matrix-matrix multiplication,  
the code below combines prefetching with tiling:

```
load first tile from global memory into registers;  
loop  
{  
    deposit tile from registers to shared memory;  
    __syncthreads();  
    load next tile from global memory into registers;  
    process current tile;  
    __syncthreads();  
}
```

The prefetching adds independent instructions between loading the data from global memory and processing the data.

# Performance Considerations

## 1 Dynamic Partitioning of Resources

- streaming multiprocessor resources
- the CUDA occupancy calculator

## 2 the Compute Visual Profiler

- getting started with `computeprof`
- analysis of the kernel `matrixMul`

## 3 Data Prefetching and Instruction Mix

- registers between global and shared memory
- maximizing instruction throughput

# throughput of arithmetic instructions

Number of operations per clock cycle per multiprocessor:

compute capability	1.x	2.0	3.5	6.0
32-bit floating-point add, multiply, multiply-add	8	32	192	64
64-bit floating-point add, multiply, multiply-add	1	16	64	4
32-bit integer add, logical operation, shift, compare	8	32	160	128
32-bit floating-point reciprocal, square root, log, exp, sine, cosine	2	4	32	32

# loop unrolling

Consider the following code snippet:

```
for(int k = 0; k < m; k++)
    C[i][j] += A[i][k]*B[k][j];
```

Counting all instructions:

- 1 loop branch instruction ( $k < m$ );
- 1 loop counter update instruction ( $k++$ );
- 3 address arithmetic instructions ( $[i][j]$ ,  $[i][k]$ ,  $[k][j]$ );
- 2 floating-point arithmetic instructions (+ and \*).

Of the 7 instructions, only 2 are floating point.

Loop unrolling reduces the number of loop branch instructions, loop counter updates, address arithmetic instructions.

**Note:** `gcc -funroll-loops` is enabled with `gcc -O2`.

## summary and exercises

We covered §6.3, §6.4, and §6.5 in the book of Kirk & Hwu;  
using data from Appendix G in the CUDA programming Guide.

- 1 Examine the occupancy calculator for the graphics card on your laptop or desktop.
- 2 Read the user guide of the compute visual profiler and perform a run on GPU code you wrote (of some previous exercise or your code for the third project). Explain the analysis of the kernel.
- 3 Redo the first “interactions between resource limitations” of this lecture using the specifications for compute capability 1.1.
- 4 Redo the second “interactions between resource limitations” of this lecture using the specifications for compute capability 1.1.