CUDA Performance Tuning Introduction

Hans Henrik Brandenborg Sørensen

DTU Computing Center

DTU Compute

<hhbs@dtu.dk>

CUDA C Best Practices Guide

1. Assessing your application.
2. Heterogeneous computing.
3. Application profiling.
4. Parallelizing your application
5. Getting started.
6. Getting the right answer.
7. Optimizing CUDA applications.
8. Performance metrics.
9. Memory optimizations.
10. Execution configuration optimizations.
11. Instruction optimizations.
12. Control flow.
13. Deploying CUDA applications.
14. Understanding the programming environment.
15. Preparing for deployment.
16. Deployment infrastructure tools.

Overview

- Recap from week 1 (now with GPUs)
  - Performance metrics
  - Measuring runtime
  - Speed-up
- Transpose tuning example
  - How many threads should I launch?
  - Latency hiding
- Identifying the performance bounds

GPU performance metrics

Performance tuning terminology

- Execution time [seconds]
  - Time to run the application (wall or cpu/gpu)
- Performance [Gflops]
  - How many floating point operations per second
- Latency [cycles or seconds]
  - Time from initiating a memory access or other action until the result is available
- Bandwidth [Gb/s]
  - The rate at which data can be transferred
- Blocking [blocksize]
  - Dividing matrices into tiles to fit memory hierarchy

You know these from week 1+2 of this course!

Performance tuning terminology

- Throughput [#/s or Gb/s]
  - Sustained rate for instructions executed or data reads + writes achieved in practice
- Occupancy [%]
  - Ratio of active warps to max possible active warps
- Instruction level parallelism (ILP) [#]
  - How many independent instructions can be executed (=pipelining)
- Thread level parallelism (TLP) [#]
  - How many independent threads can be launched
- Coalescing
  - All threads in a warp are reading data from a contiguous, aligned, region of global memory

Common GPU optimization teminology.

Measuring runtime for kernels

- Using CPU timers
  - Run your kernel several times and compute average
  - Remember that kernel launches are non-blocking
    - Add `cudaDeviceSynchronize()`
  - GPUs have a “wake-up” time to create CUDA context

Measuring runtime for kernels

- Using CPU timers
  - Run your kernel several times and compute average
  - Remember that kernel launches are non-blocking
    - Add `cudaDeviceSynchronize()`
  - GPUs have a “wake-up” time to create CUDA context
- E.g., use the OpenMP timer

```
#include <omp.h>

double time = omp_get_wtime();

kernelFunc<<<dimGrid, dimBlock>>>();
cudaDeviceSynchronize();

double elapsed = omp_get_wtime()-time;
```

Measuring runtime for kernels

- Using CUDA GPU timers

```
cudaEvent_t start, stop;
float elapsed;

cudaEventCreate(&start); cudaEventCreate(&stop);

cudaEventRecord(start, 0);
kernelFunc<<<dimGrid, dimBlock>>>();
cudaEventRecord(stop, 0);
cudaEventSynchronize(stop);
cudaEventElapsedTime(&elapsed, start, stop);

cudaEventDestroy(start); cudaEventDestroy(stop);
```

Events should be used, e.g., if you want separate runtimes of several kernels running simultaneously

Speed-up (now with GPUs)

- GPU definition of speed-up is traditionally

$$\texttt{Speedup} = \frac{\texttt{CPUtime}[\mathrm{s}]}{\texttt{GPUtime}[\mathrm{s}]}$$

  and usually written in times ($\times$) manner, e.g., 3.2$\times$

- Useful for indicating performance without telling what the performance actually is(!)

Speed-up (now with GPUs)

- GPU definition of speed-up is traditionally

$$\texttt{Speedup} = \frac{\texttt{CPUtime}[\mathrm{s}]}{\texttt{GPUtime}[\mathrm{s}]}$$

  and usually written in times ($\times$) manner, e.g., 3.2$\times$

- Useful for indicating performance without telling what the performance actually is(!)

But be fair when comparing to CPU times – speed-ups of 100$\times$-1000$\times$ (see Nvidia Show-case homepage) are unrealistic when the hardware specs are taken into account

Which performance metric to use?

- Compute bound
  - Limited by # flops * time per flop

Gflops = # floating point operations / 10^9 / runtime

Which performance metric to use?

- Compute bound
  - Limited by # flops * time per flop

Gflops = # floating point operations / 10^9 / runtime

- Memory bound
  - Limited by # bytes moved / bandwidth

Bandwidth = (Bytes_read + Bytes_written) / 10^9 / runtime

How to assess your performance?

- Compute bound
  - Compare with the theoretical peak performance
    - Run device query to find specs and calculate, e.g.

SP: 2880 cores * 0.745 GHz * 2 flops per core = 4291 Gflops

DP: (1/3) * 4291 Gflops = 1430 Gflops

How to assess your performance?

- Compute bound
  - Compare with the theoretical peak performance
    - Run device query to find specs and calculate, e.g.

SP: 2880 cores * 0.745 GHz * 2 flops per core = 4291 Gflops

DP: (1/3) * 4291 Gflops = 1430 Gflops

- Memory bound
  - Compare with the theoretical peak bandwidth
    - Run device query to find specs and calculate, e.g.

Peak bandwidth = 3.004 GHz * (384 / 8) bytes * 2 = 288 GB/s

How to assess your performance?

- Compute bound
  - Compare with the theoretical peak performance
    - Run device query to find specs and calculate, e.g.

SP: 2880 cores e = 4291 Gflops

DP: s = 1430 Gflops

40-60%: okay
60-75%: good
>75%: excellent

- Memory b
  - Compare with the theoretical peak bandwidth
    - Run device query to find specs and calculate, e.g.

Peak bandwidth = 3.004 GHz * (384 / 8) bytes * 2 = 288 GB/s

How to assess your speed-up?

- Find the CPU specs online: http://ark.intel.com/products/75790/Intel-Xeon-Processor-E5-2630-v2-15M-Cache-2_60-GHz

How to assess your speed-up?

- Find the CPU specs online: http://ark.intel.com/products/75790/Intel-Xeon-Processor-E5-2630-v2-15M-Cache-2_60-GHz

SP: 6 cores * 2.6 GHz * 8 (AVX) * 2 flops per core = 249 Gflops

DP: (1/2) * 249 Gflops = 124 Gflops

Peak bandwidth = 51.2 GB/s

How to assess your speed-up?

- Find the CPU specs online: http://ark.intel.com/products/75790/Intel-Xeon-Processor-E5-2630-v2-15M-Cache-2_60-GHz

SP: 6 cores * 2.6 GHz * 8 (AVX) * 2 flops per core = 249 Gflops

DP: (1/2) * 249 Gflops = 124 Gflops

Peak bandwidth = 51.2 GB/s

- Expected speed-up

	1 CPU	2 CPUs
Compute bound: 1430/124 =	11.5x	5.8x
Memory bound: 288/51.2 =	5.6x	2.8x

How to assess your speed-up?

- Find the CPU specs online: http://ark.intel.com/products/75790/Intel-Xeon-Processor-E5-2630-v2-15M-Cache-2_60-GHz

SP: 6 cores * 2.6 ... core = 249 Gflops

DP: ... ops = 124 Gflops

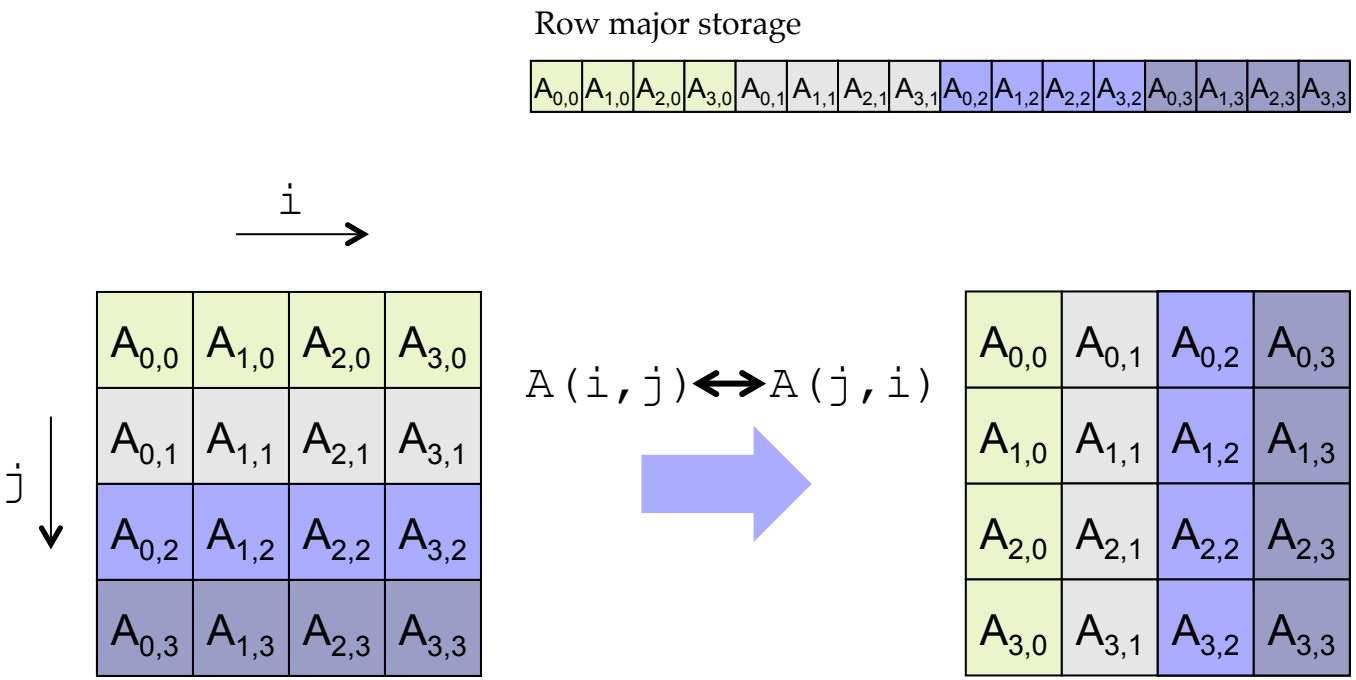
40-60%: okay
60-75%: good
>75%: excellent

- Expected speed-up

	1 CPU	2 CPUs
Compute bound: 1430/124 =	11.5x	5.8x
Memory bound: 288/51.2 =	5.6x	2.8x

Transpose example

Transpose example



- We will use this example to illustrate the process of performance tuning a CUDA kernel “step-by-step”

Transpose example (v1 serial)

```
// Reference serial CPU transpose
__host__ __device__
void transpose(double *A, double *At)
{
   for(int j=0; j < N; j++)
      for(int i=0; i < N; i++)
         At[j + i*N] = A[i + j*N]; // At(j,i)=A(i,j)
}
```

```
// Kernel to be launched on a single thread
__global__
void transpose_serial(double *A, double *At)
{
    transpose(A, At);
}
```

Transpose example (v1 serial)

```
#define N 2880
...
   // CPU reference transpose for checking result
   transpose(h_A, h_At_CPU);

   // Transfer matrix to device
   cudaMemcpy(d_A, h_A, A_size, cudaMemcpyHostToDevice);

   transpose_serial<<<1, 1>>>(d_A, d_At);
   cudaDeviceSynchronize();

   // Transfer result to host
   cudaMemcpy(h_At, d_At, A_size, cudaMemcpyDeviceToHost);
...
```

Transpose example (v1 serial)

```
$ nvprof --print-gpu-summary ./transpose_gpu
==30836== Profiling application: ./transpose_gpu
==30836== Profiling result:
Time(%)      Time     Calls       Avg       Min       Max  Name
 98.10%  2.52254s         1  2.52254s  2.52254s  2.52254s  transpose_serial(double*, double*)
  1.08%  27.837ms         1  27.837ms  27.837ms  27.837ms  [CUDA memcpy DtoH]
  0.80%  20.670ms         1  20.670ms  20.670ms  20.670ms  [CUDA memcpy HtoD]
  0.01%  331.49us         1  331.49us  331.49us  331.49us  [CUDA memset]
$
```

Version	v1 serial
Time [ms]	2523

Transpose example (v2 per row)

```
// Kernel to be launched with one thread per row of A
__global__
void transpose_thread_per_row(double *A, double *At)
{
   // Thread index gives row of A
   int j = blockIdx.x * blockDim.x + threadIdx.x;

   for(int i=0; i < N; i++)
      At[j + i*N] = A[i + j*N]; // At(j,i)=A(i,j)
}
```

```
...
   transpose_per_row<<<15, 192>>>(d_A, d_At);
...
```

Transpose example (v2 per row)

```
$ nvprof --print-gpu-summary ./transpose_gpu
==32362== Profiling application: ./transpose_gpu
==32362== Profiling result:
Time(%)      Time     Calls       Avg       Min       Max  Name
 90.04%  438.38ms       100  4.3838ms  4.2700ms  4.7287ms  transpose_per_row(double*, double*)
  5.68%  27.679ms         1  27.679ms  27.679ms  27.679ms  [CUDA memcpy DtoH]
  4.21%  20.499ms         1  20.499ms  20.499ms  20.499ms  [CUDA memcpy HtoD]
  0.07%  330.34us         1  330.34us  330.34us  330.34us  [CUDA memset]
$
```

Version	v1 serial	v2 per row
Time [ms]	2523	4.38

Transpose example (v3 per elm)

```
// Kernel to be launched with one thread per element of A
__global__
void transpose_per_elm(double *A, double *At)
{
   // 2D thread indices defining row and col of element
   int j = blockIdx.x * blockDim.x + threadIdx.x;
   int i = blockIdx.y * blockDim.y + threadIdx.y;

   At[j + i*N] = A[i + j*N]; // At(j,i)=A(i,j)
}
```

```
#define K 16
...
   transpose_per_elm<<<dim3(N/K, N/K), dim3(K,K)>>>(d_A,
d_At);
...
```

Transpose example (v3 per elm)

```
$ nvprof --print-gpu-summary ./transpose_gpu
==3324== Profiling application: ./transpose_gpu
==3324== Profiling result:
Time(%)      Time     Calls       Avg       Min       Max  Name
 74.88%  153.66ms       100  1.5366ms  1.5283ms  1.5410ms  transpose_per_elm(double*, double*)
 13.60%  27.914ms         1  27.914ms  27.914ms  27.914ms  [CUDA memcpy DtoH]
 11.36%  23.310ms         1  23.310ms  23.310ms  23.310ms  [CUDA memcpy HtoD]
  0.16%  331.20us         1  331.20us  331.20us  331.20us  [CUDA memset]
$
```

Version	v1 serial	v2 per row	v3 per elm
Time [ms]	2523	4.38	1.53

How many threads should I run?

- Why is one thread per CUDA core not enough?
  - E.g. `kernelFunc<<<15, 192>>>(...);`

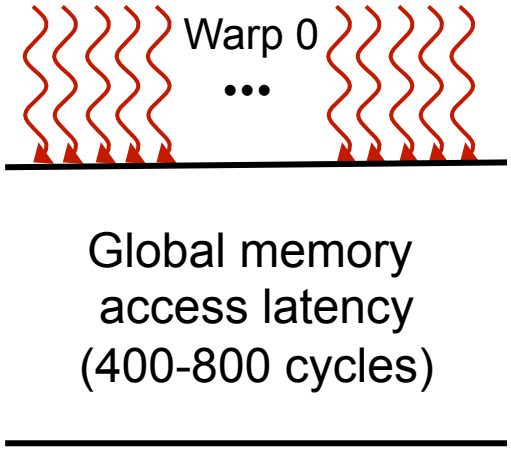
How many threads should I run?

- Why is one thread per CUDA core not enough?
  - E.g. `kernelFunc<<<15, 192>>>(...);`

How many threads should I run?

- Why is one thread per CUDA core not enough?
  - E.g. `kernelFunc<<<15, 192>>>(...);`

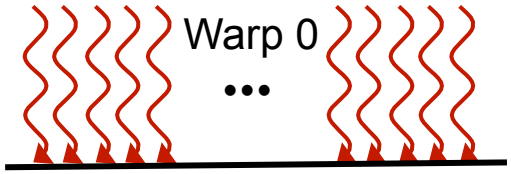
```
A[i + j*N];
```



How many threads should I run?

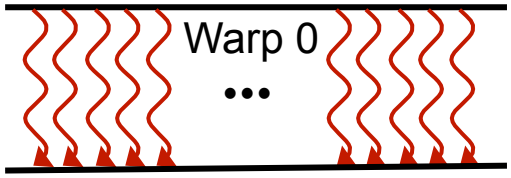
- Why is one thread per CUDA core not enough?
  - E.g. `kernelFunc<<<15, 192>>>(...);`

```
A[i + j*N];
```



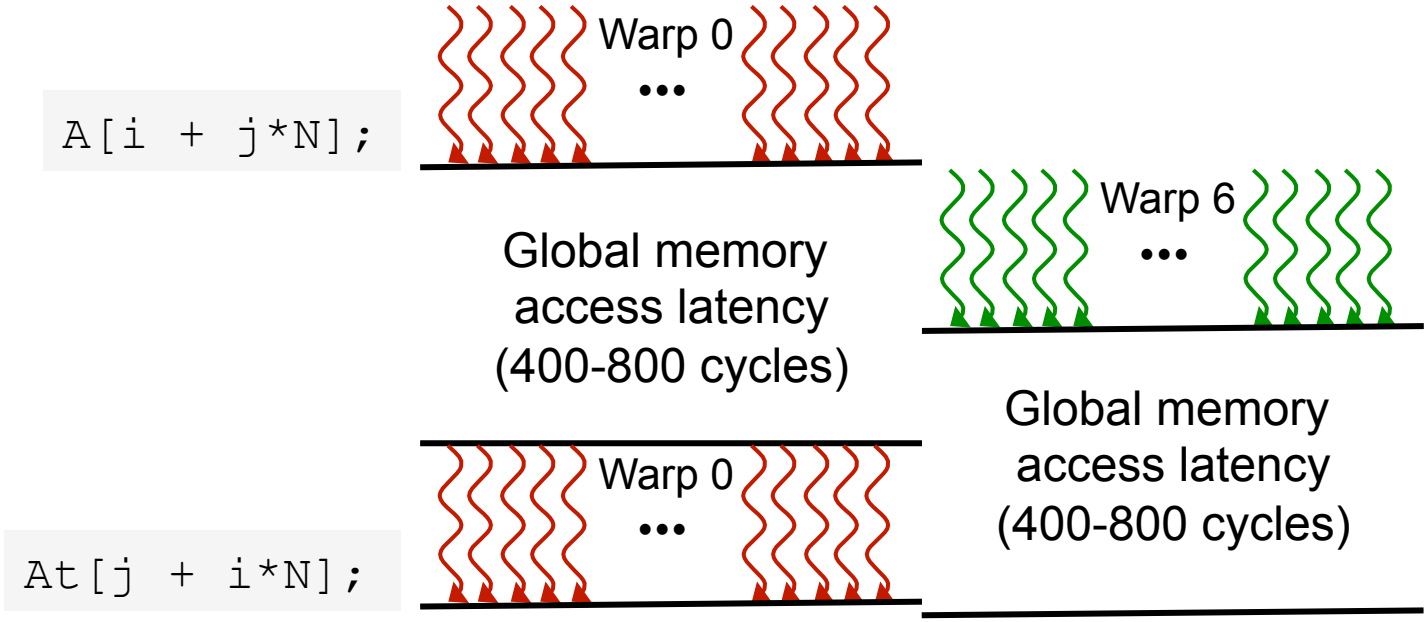
Global memory
access latency
(400-800 cycles)

```
At[j + i*N];
```



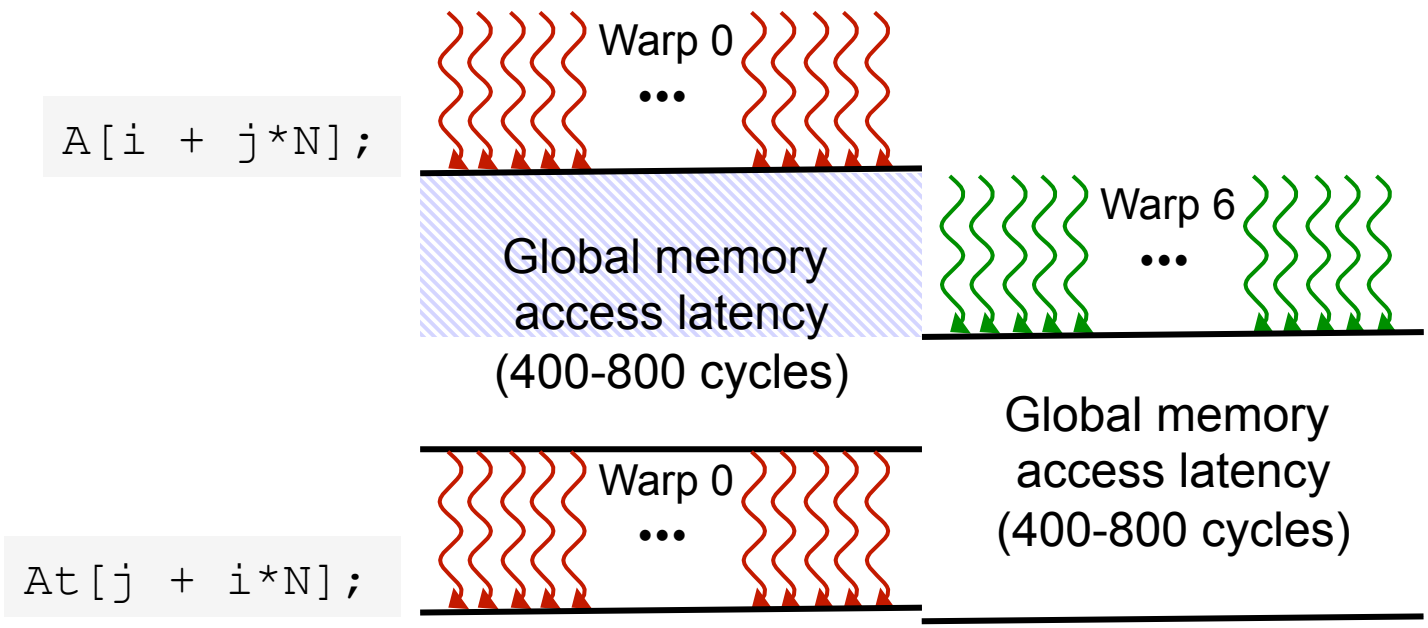
How many threads should I run?

- Why is one thread per CUDA core not enough?
  - E.g. `kernelFunc<<<15, 192>>>(...);`



How many threads should I run?

- Why is one thread per CUDA core not enough?
  - E.g. `kernelFunc<<<15, 192>>>(...);`



- Reason: We can hide memory latency by having idle warps to schedule while waiting for data

Identifying Performance Bounds

Compute bound or memory bound?

- Perfect balance / Arithmetic intensity
  - Every GPU has its own perfect instructions-to-bytes ballance
  - E.g., Fermi C2050's perfect ratio ~4.5 : 1 with ECC on
  - If we are higher = compute bound; lower = memory bound

Compute bound or memory bound?

- Perfect balance / Arithmetic intensity
  - Every GPU has its own perfect instructions-to-bytes ballance
  - E.g., Fermi C2050's perfect ratio ~4.5 : 1 with ECC on
  - If we are higher = compute bound; lower = memory bound

- Rough algorithmic analysis
  - How many instructions needed, how many bytes
  - E.g. matrix-vector multiplication (square matrix of size N)
    - $2*N^2$ Flops : $8*(N^2+2*N)$ Bytes ➔ memory bound
  - E.g. matrix-matrix multiplication (square matrices of size N)
    - $2*N^3$ Flops : $8*(3*N^2)$ Bytes ➔ compute bound

  - For simple kernels, you will usually not be in doubt!

Analysis of performance bounds

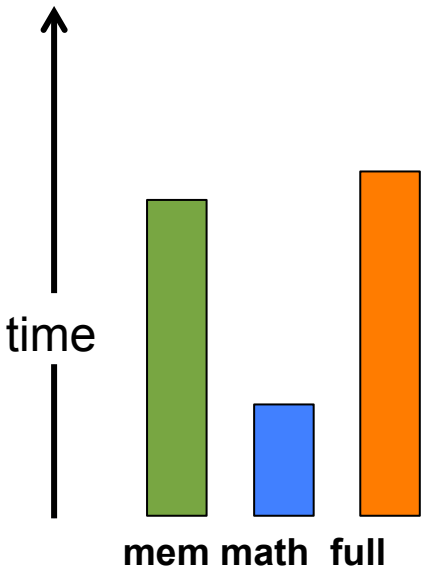
- Using a profiler
  - Instruction count, memory request/transaction count, timings, throughtput etc.
  - Fast and reliable + easy to use
  - Many counters and plots - we will get back to these + examples

- We will use the profiler `nvvp`
  - More on this tomorrow

Analysis of performance bounds

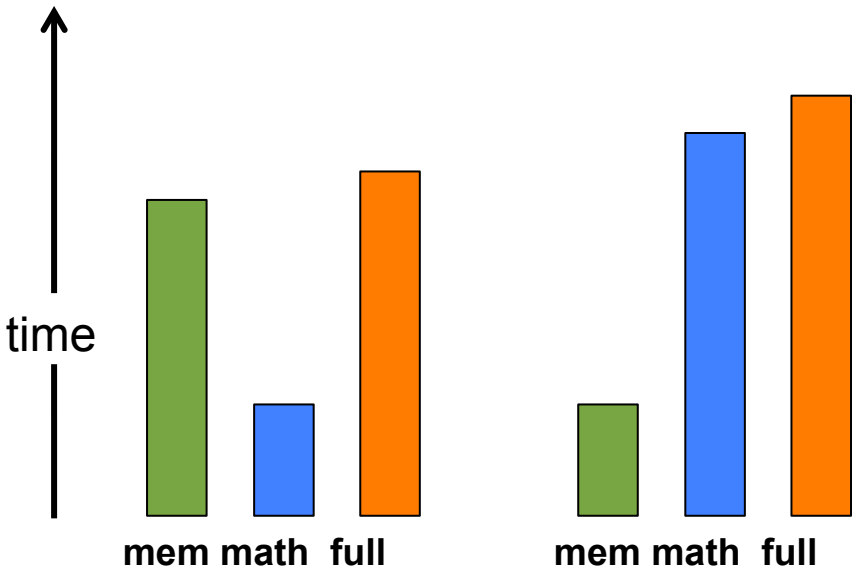
- Modifying source code
  - Comment out arithmetic ➔ memory-only version
  - Make sure the memory access pattern + #registers is the same
  - Comment out memory accesses ➔ math-only version
  - Trick the compiler to keep it from optimizing code away
    - Put writes inside `if {...}` that always evaluates to `false`
  - Gives you good estimates for where time is spend
  - More accurate than with profiler, but requires work

Possible performance scenarios



Memory-bound
Good mem-math overlap
(assuming memory throughput is not low compared to HW theory)

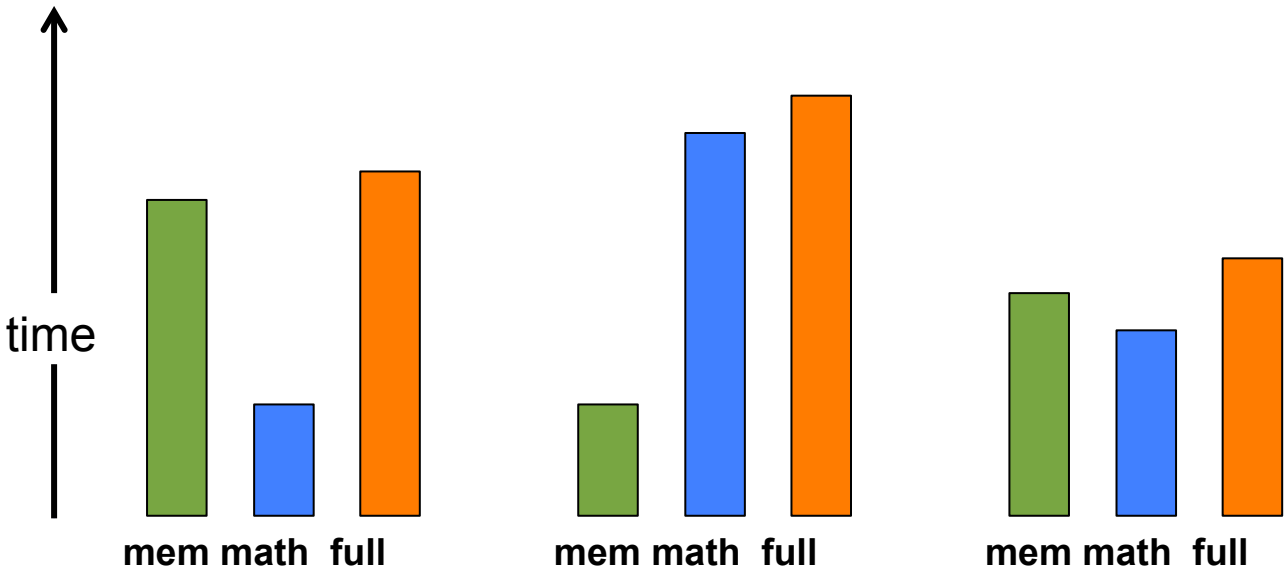
Possible performance scenarios



Memory-bound
Good mem-math overlap
(assuming memory throughput is not low compared to HW theory)

Compute-bound
Good mem-math overlap
(assuming instr. throughput is not low compared to HW theory)

Possible performance scenarios

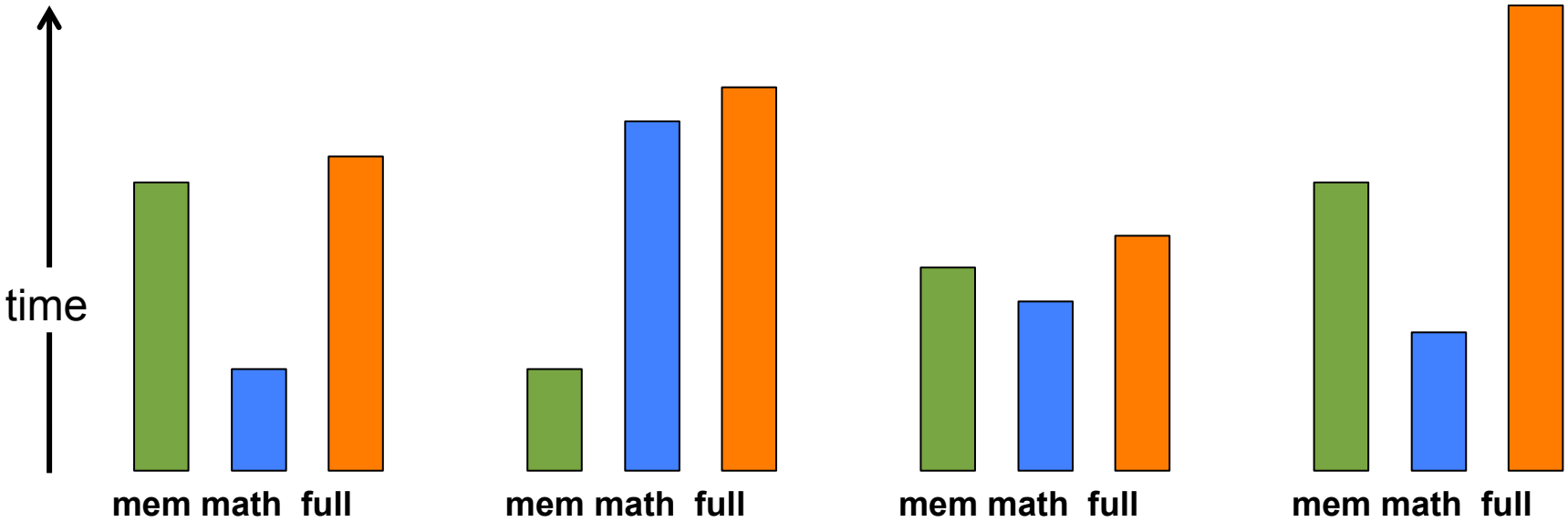


Memory-bound
Good mem-math overlap
(assuming memory throughput is not low compared to HW theory)

Compute-bound
Good mem-math overlap
(assuming instr. throughput is not low compared to HW theory)

Balanced
Good mem-math overlap
(assuming memory and instruction throughput is not low compared to HW theory)

Possible performance scenarios



Memory-bound
Good mem-math overlap
(assuming memory throughput is not low compared to HW theory)

Compute-bound
Good mem-math overlap
(assuming instr. throughput is not low compared to HW theory)

Balanced
Good mem-math overlap
(assuming memory and instruction throughput is not low compared to HW theory)

Occupancy/memory bound
Poor mem-math overlap: latency is a problem

CUDA performance tuning process

- Determine what limits kernel performance
  - Parallelism (concurrency bound)
  - Memory accesses (memory bound)
  - Floating point operations (compute bound)

… or a combination

- Use appropriate performance metric for the kernel
  - Or use speed-up between kernel modifications
- Address the limiters in the order of importance
  - Determine how close to the theoretical peaks
  - Analyze
  - Apply optimizations

… and iterate with small steps

End of lecture