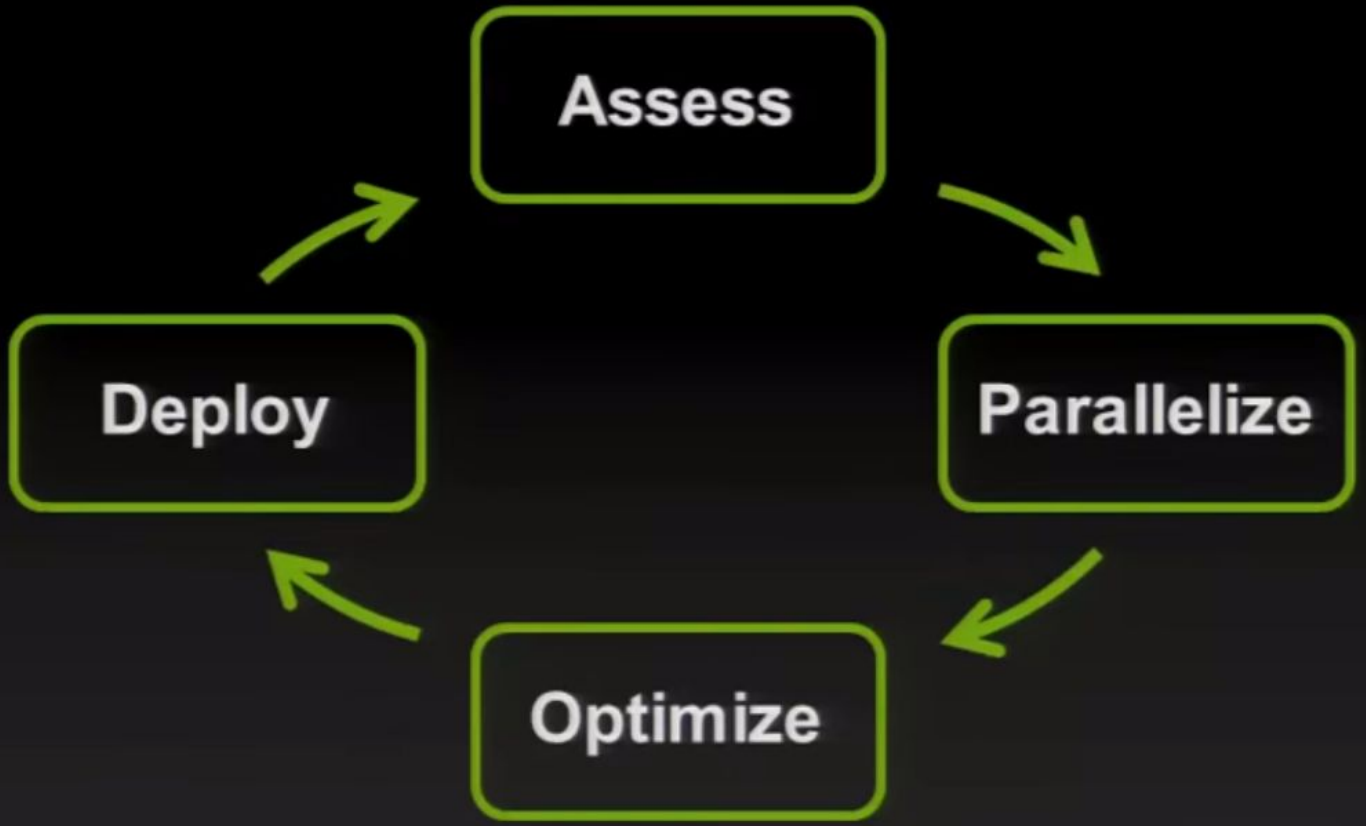
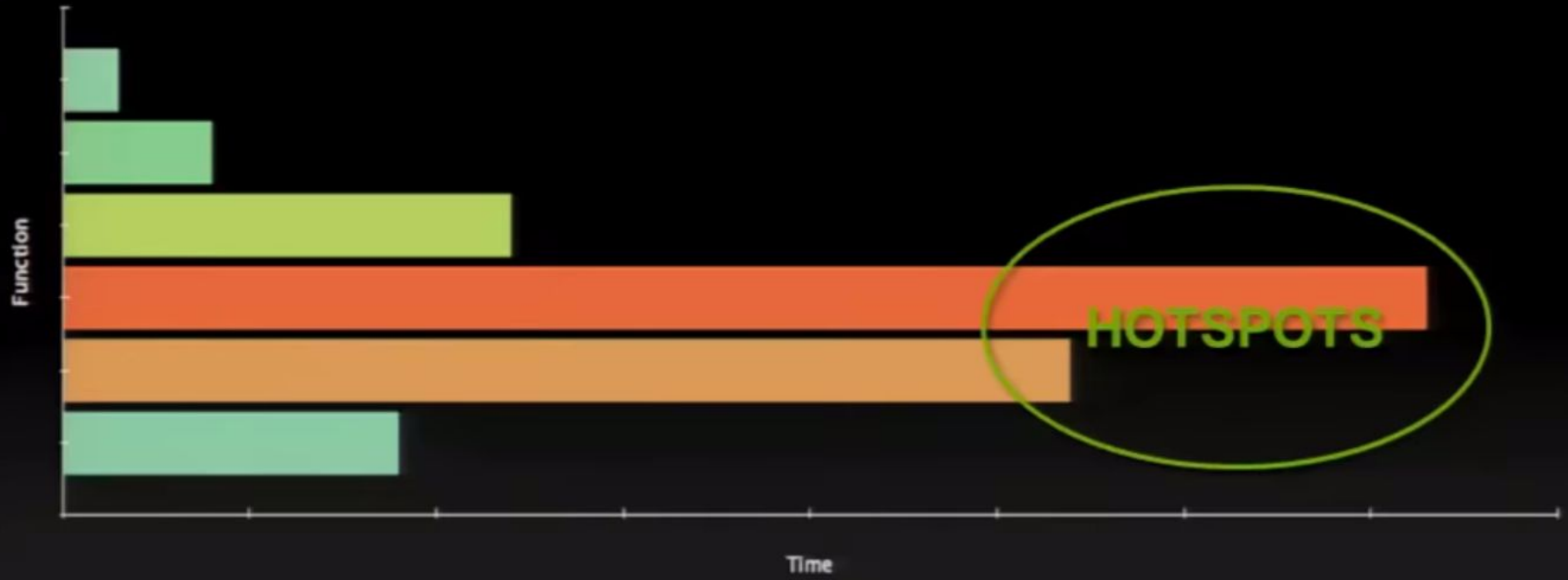
CUDA Optimizations - 1

COL 730 class 3

APOD: A Systematic Path to Performance



Assess



- Identify hotspots (total time, number of calls)
- Understand scaling (strong and weak)

Optimize

- Profile-driven optimization
- Tools:
  - nsight Visual Studio Edition or Eclipse Edition
  - nvvp NVIDIA Visual Profiler
  - nvprof Command-line profiling

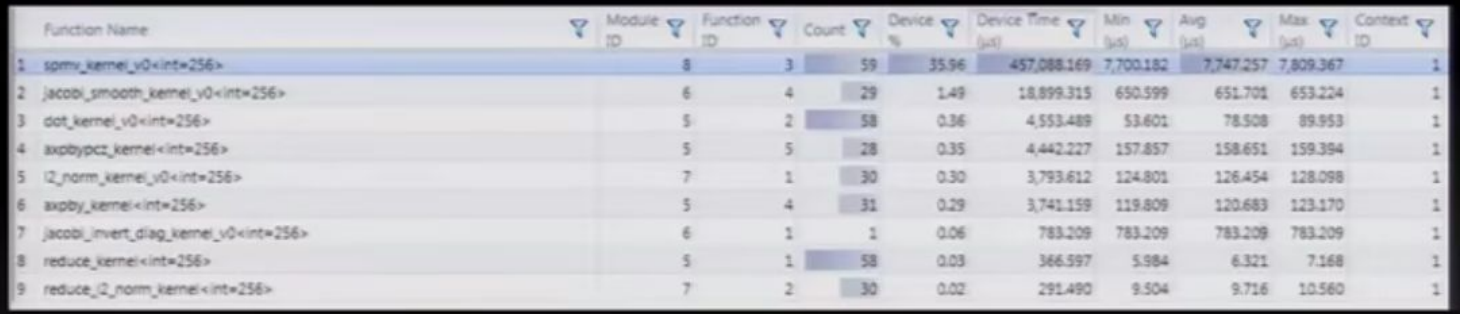
Deploy

Productize

- Check API return values
- Run cuda-memcheck tools
- Library distribution
- Cluster management

Early gains
Subsequent changes are evolutionary

Assess



	Function Name	Module ID	Function ID	Count	Device %	Device Time (µs)	Min (µs)	Avg (µs)	Max (µs)	Context ID
1	somv_kernel_v0<int=256>	8	3	59	35.96	457,088.169	7,700.182	7,747.257	7,809.367	1
2	jacobi_smooth_kernel_v0<int=256>	6	4	29	1.49	18,899.315	650.599	651.701	653.224	1
3	dot_kernel_v0<int=256>	5	2	58	0.36	4,553.489	53.601	78.508	89.953	1
4	axpbypcz_kernel<int=256>	5	5	28	0.35	4,442.227	157.857	158.651	159.384	1
5	l2_norm_kernel_v0<int=256>	7	1	30	0.30	3,793.612	124.801	126.454	128.098	1
6	axpby_kernel<int=256>	5	4	31	0.29	3,741.159	119.809	120.683	123.170	1
7	jacobi_invert_diag_kernel_v0<int=256>	6	1	1	0.06	783.209	783.209	783.209	783.209	1
8	reduce_kernel<int=256>	5	1	58	0.03	366.597	5.984	6.321	7.168	1
9	reduce_l2_norm_kernel<int=256>	7	2	30	0.02	291.490	9.504	9.716	10.560	1

- Profile the code, find the hotspot(s)
- Focus your attention where it will give the most benefit

Assess

- We've found a hotspot to work on!
  - What percent of our total time does this represent?
  - How much can we improve it? What is the "speed of light"?
  - How much will this improve our overall performance?

Assess

- Let's investigate…
  - Strong scaling and Amdahl's Law
  - Weak scaling and Gustafson's Law
  - Expected perf limiters: Bandwidth? Computation? Latency?

Assess: Understanding Scaling

Strong Scaling

- A measure of how, for fixed overall problem size, the time to solution decreases as more processors are added to a system
- Linear strong scaling: speedup achieved is equal to number of processors used

- Amdahl's Law:

$$S = \frac{1}{(1-P) + \frac{P}{N}} \approx \frac{1}{(1-P)}$$

Assess: Understanding Scaling

Weak Scaling

- A measure of how time to solution changes as more processors are added with fixed problem size *per processor*
- Linear weak scaling: overall problem size increases as num. of processors increases, but execution time remains constant

- Gustafson's Law:

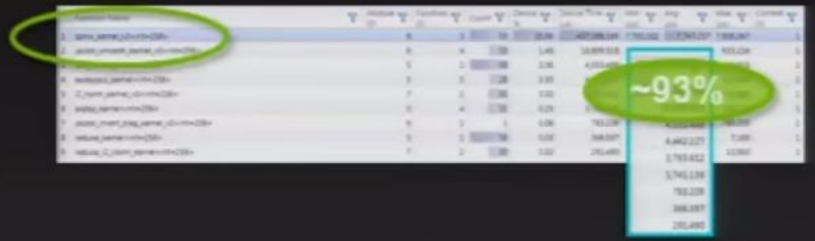
$$S = N + (1 - P)(1 - N)$$

Assess: Applying Strong and Weak Scaling

- Understanding which type of scaling is most applicable is an important part of estimating speedup:
  - Sometimes problem size will remain constant
  - Other times problem size will grow to fill the available processors
- Apply either Amdahl's or Gustafson's Law to determine an upper bound for the speedup

Assess: Applying Strong Scaling

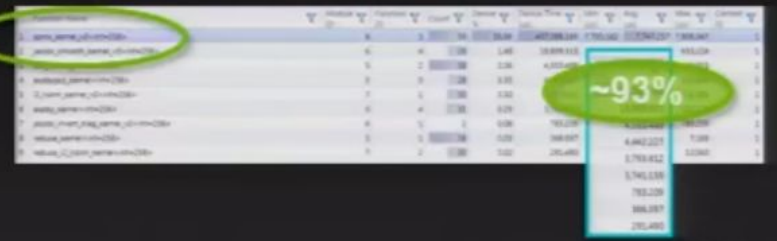
- Recall that in this case we are wanting to optimize an existing kernel with a pre-determined workload
- That's strong scaling, so Amdahl's Law will determine the maximum speedup



Assess: Applying Strong Scaling

Say, for example, our kernel is ~93% of total time:

- Speedup $S = \frac{1}{(1-P)+\frac{P}{S_P}}$ (S_P = speedup in parallel part)
- In the limit when S_P is huge, S will approach $\frac{1}{1-0.93} \approx 14.3\times$
- In practice, it will be less than that depending on the S_P achieved
- Getting S_P to be high is the goal of optimizing, of course



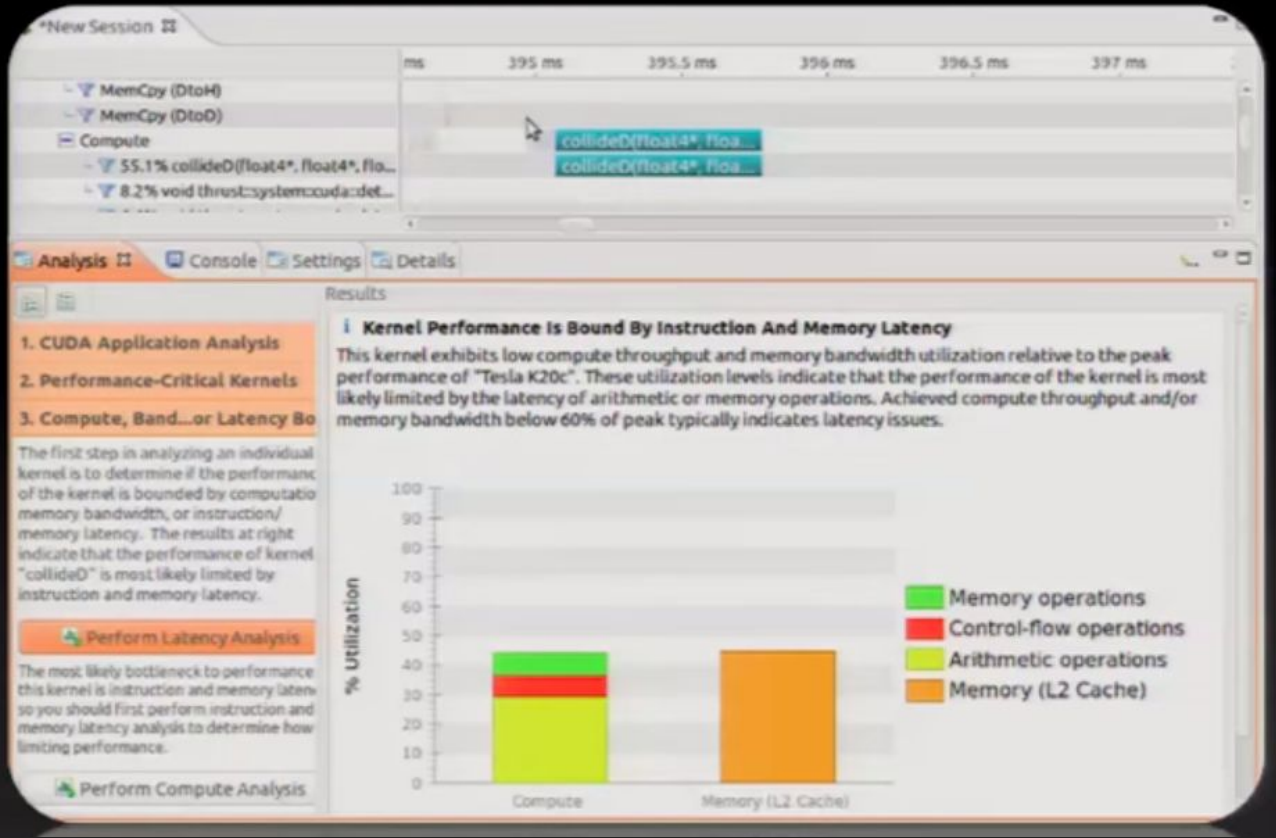
Assess: Speed of Light

- What's the limiting factor?
  - Memory bandwidth?
  - Compute throughput?
  - Latency?
  - Not sure?
    - Get a rough estimate by counting bytes per instruction, compare it to "balanced" peak ratio $\frac{GBytes/sec}{Ginsns/sec}$
    - Profiler will help you determine this

Assess: Limiting Factor

- Comparing bytes per instr. will give you a guess as to whether you're likely to be bandwidth-bound or instruction-bound
- Comparing actual achieved GB/s vs. theory and achieved Ginstr/s vs. theory will give you an idea of how well you're doing
  - If both are low, then you're probably latency-bound and need to expose more (concurrent) parallelism

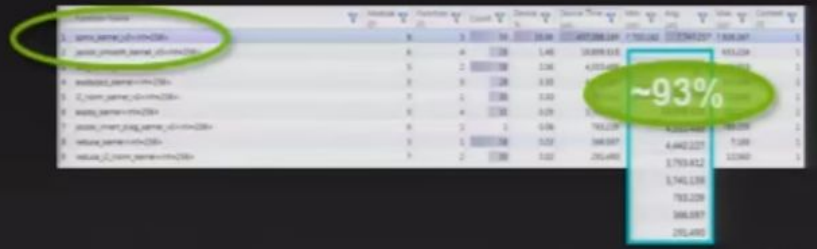
Assess: Limiting Factor



Assess: Speed of Light

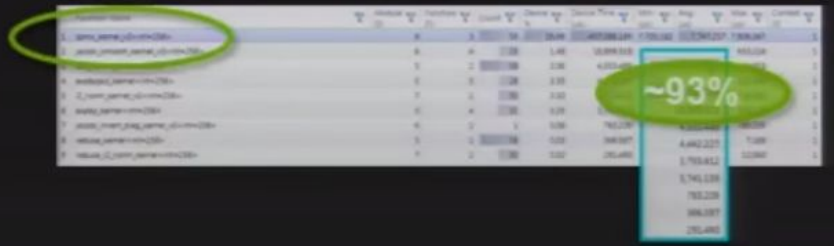
- What's the limiting factor?
  - Memory bandwidth? Compute throughput? Latency?
- Consider SpMV: intuitively expect it to be bandwidth-limited
  - Say we discover we're getting only ~38% of peak bandwidth
  - If we aim to get this up to ~65% of peak, that's 1.7× for this kernel
  - 1.7× for this kernel translates into 1.6× overall due to Amdahl:

$$S = \frac{1}{(1-0.93)+\frac{0.93}{1.7}} \approx 1.6\times$$



Assess: Limiting Factor

- For our example SpMV kernel, our first discovery was that we're latency-limited, not bandwidth, since utilization was so low
- This tells us our first "optimization" step actually needs to be related how we expose (memory-level) parallelism



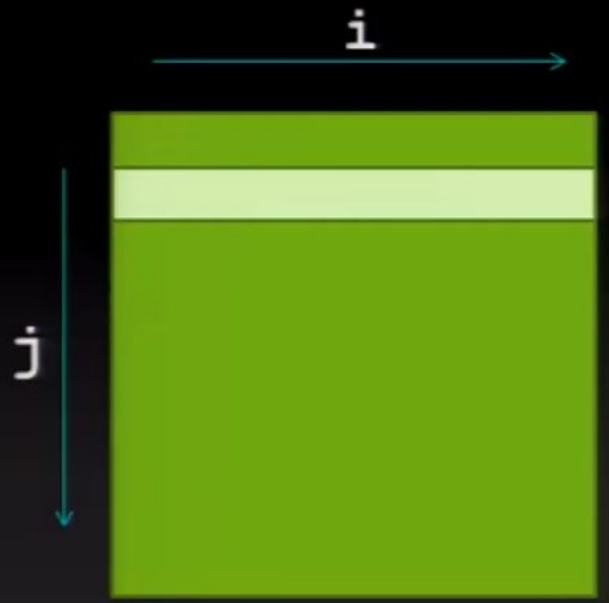
Parallelize

Parallelism Needed

- GPU is a parallel machine
  - Lots of arithmetic pipelines
  - Multiple memory banks
- To get good performance, your code must expose sufficient parallelism for 2 reasons:
  - To actually give work to all the pipelines
  - To hide latency of the pipelines
- Rough rule of thumb for Tesla K20X:
  - You want to have 14K or more threads running concurrently

Case Study: Matrix Transpose

```
void transpose(float in[][], float out[][], int N)
{
  for(int j=0; j < N; j++)
    for(int i=0; i < N; i++)
      out[j][i] = in[i][j];
}
```



An Initial CUDA Version

```
__global__ void transpose(float in[], float out[], int N)
{
  for(int j=0; j < N; j++)
     for(int i=0; i < N; i++)
       out[i*N+j] = in[j*N+i];

}


float in[N*N], out[N*N];
…
transpose<<<1,1>>>(in, out, N);
```

+ Quickly implemented

- Performance weak

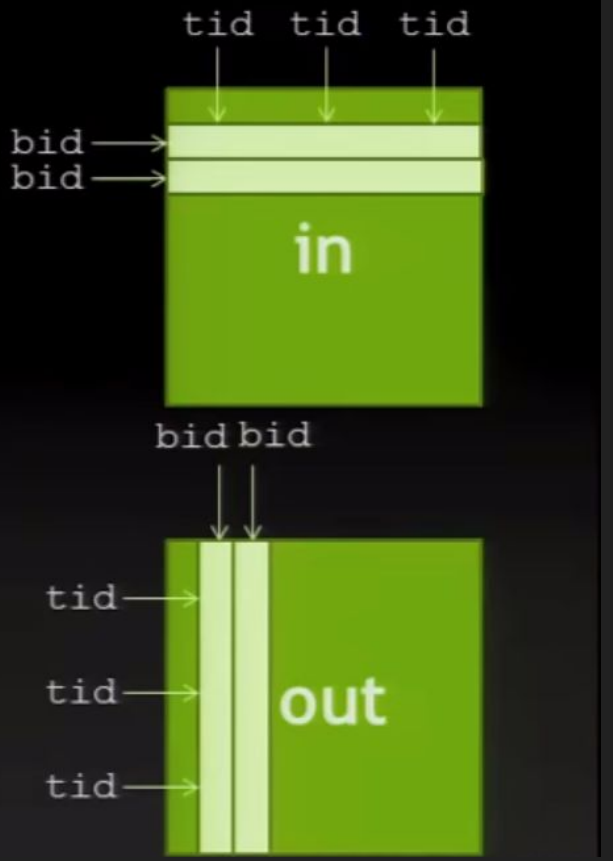
Parallelize across matrix elements

Process elements independently

```
__global__ transpose(float in[], float out[])
{
  int tid = threadIdx.x;
  int bid = blockIdx.x;

  out[tid*N+bid] = in[bid*N+tid];
}


float in[], out[];
…
transpose<<<N,N>>>(in, out);
```



Kernel Launch Configuration

- A kernel is a function that runs on the GPU
- A kernel is launched as a grid of blocks of threads
- Launch configuration is the number of blocks and number of threads per block, expressed in CUDA with the <<< >>> notation:

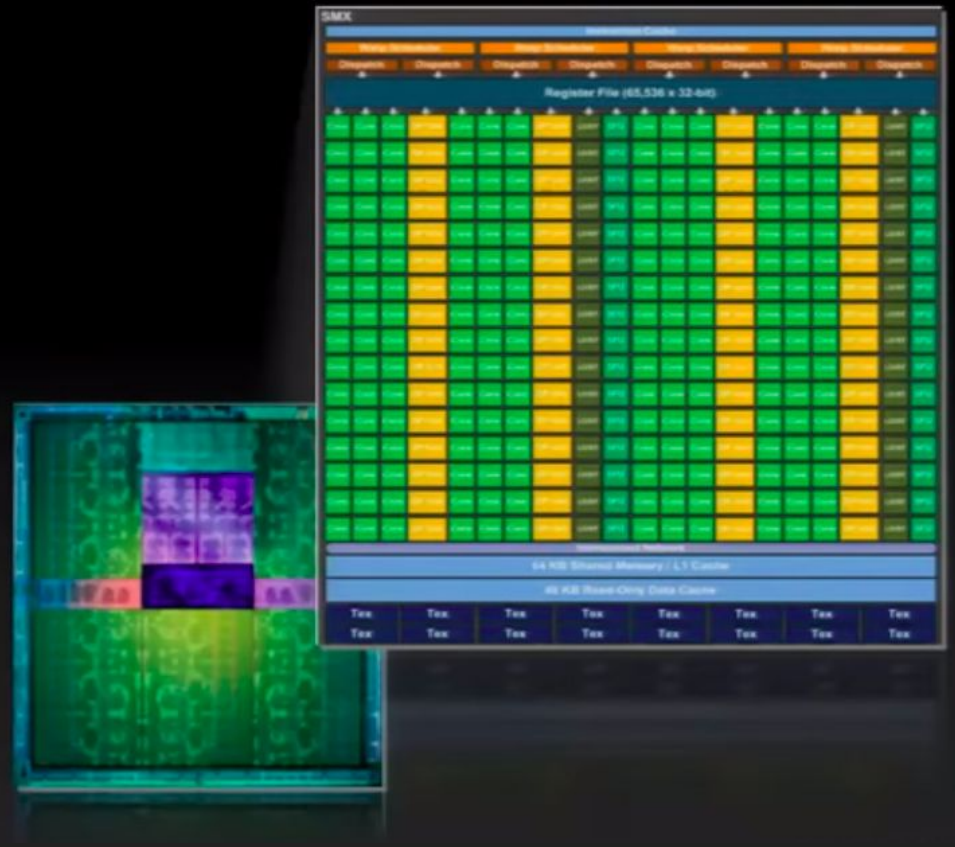
```
mykernel<<<blocks_per_grid,threads_per_block>>>(…);
```

- What values should we pick for these?
  - Need enough total threads to process entire input
  - Need enough threads to keep the GPU busy
  - Selection of block size is an optimization step involving *warp occupancy*

Kepler Streaming Multiprocessor (SMX)

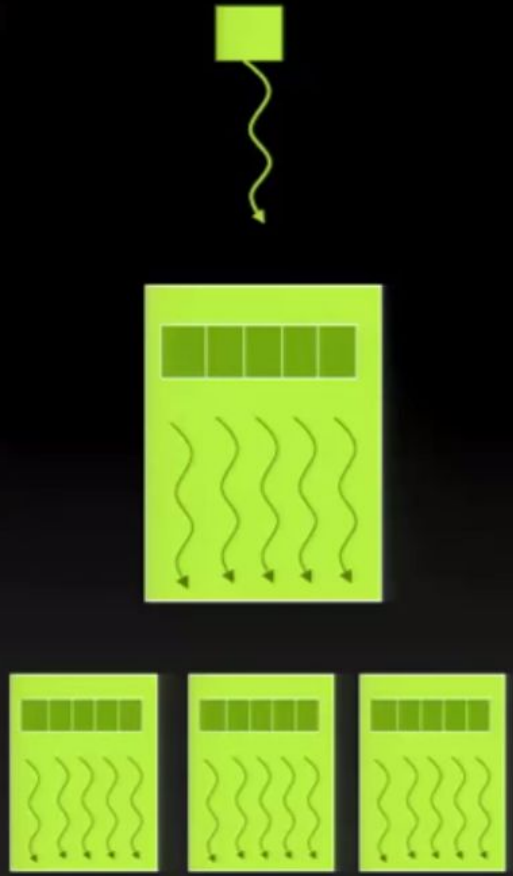
Per SMX:

- 192 SP CUDA Cores
- 64 DP CUDA Cores
- 4 warp schedulers
  - Up to 2048 concurrent threads
  - One or two instructions issued per scheduler per clock from a single warp
- Register file (256KB)
- Shared memory (48KB)



CUDA Execution Model

- Thread: Sequential execution unit
  - All threads execute same sequential program
  - Threads execute in parallel
- Threads Block: a group of threads
  - Executes on a single Streaming Multiprocessor (SM)
  - Threads within a block can cooperate
    - Light-weight synchronization
    - Data exchange
- Grid: a collection of thread blocks
  - Thread blocks of a grid execute across multiple SMs
  - Thread blocks do not synchronize with each other
  - Communication between blocks is expensive



Execution Model

Software

Thread

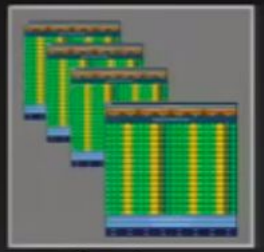
Thread Block

Grid

Hardware

CUDA Core

Multiprocessor



Device

Threads are executed by scalar CUDA Cores

Thread blocks are executed on multiprocessors

Thread blocks do not migrate

Several concurrent thread blocks can reside on one multiprocessor - limited by multiprocessor resources (shared memory and register file)

A kernel is launched as a grid of thread blocks

Warps

A thread block consists of *warps of* 32 threads

A warp is executed physically in parallel on some multiprocessor.

Threads of a warp issue instructions in lock-step (as with SIMD)

Launch Configuration: General Guidelines

How many blocks should we use?

- 1,000 or more thread blocks is best
  - Rule of thumb: enough blocks to fill the GPU at least 10s of times over
  - Makes your code ready for several generations of future GPUs

Launch Configuration: General Guidelines

How many threads per block should we choose?

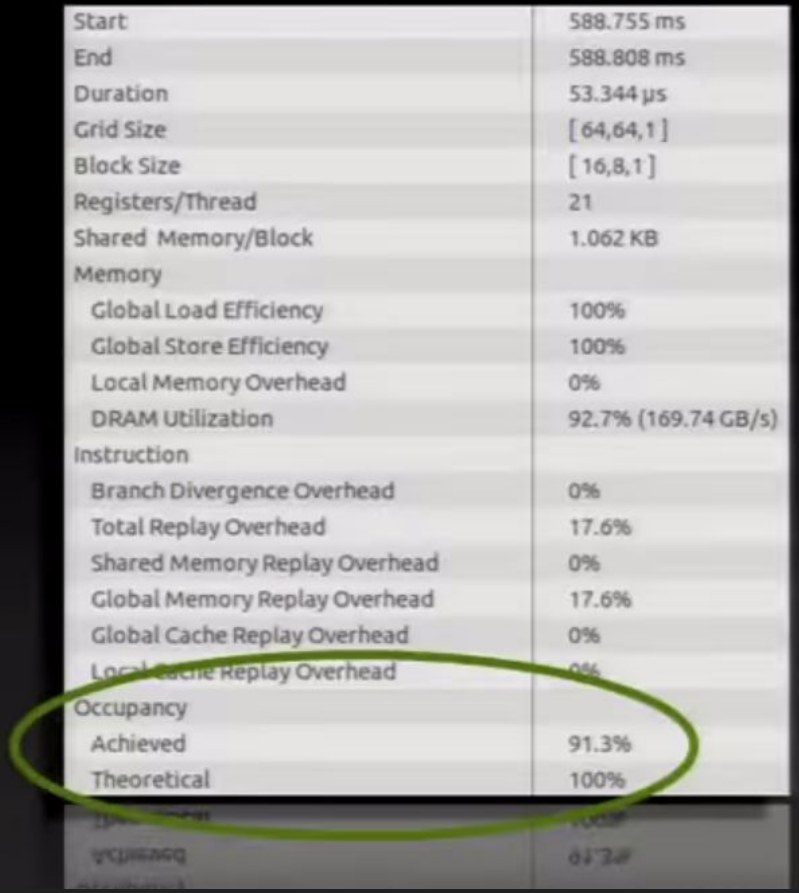
- The really short answer: 128, 256, or 512 are often good choices

- The slightly longer answer:
  - Pick a size that suits the problem well
  - Multiples of 32 threads are best
  - Pick a number of threads per block (and a number of blocks) that is sufficient to keep the SM busy

Occupancy

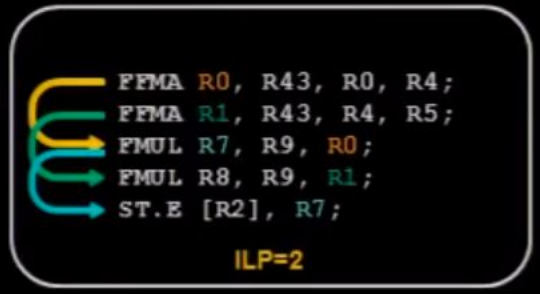
- Need enough concurrent warps per SM to hide latencies:
  - Instruction latencies
  - Memory access latencies
- Hardware resources determine number of warps that fit per SM

$$\text{Occupancy} = N_{actual} / N_{max}$$



Start	588.755 ms
End	588.808 ms
Duration	53.344 µs
Grid Size	[64,64,1]
Block Size	[16,8,1]
Registers/Thread	21
Shared Memory/Block	1.062 KB
Memory	
Global Load Efficiency	100%
Global Store Efficiency	100%
Local Memory Overhead	0%
DRAM Utilization	92.7% (169.74 GB/s)
Instruction	
Branch Divergence Overhead	0%
Total Replay Overhead	17.6%
Shared Memory Replay Overhead	0%
Global Memory Replay Overhead	17.6%
Global Cache Replay Overhead	0%
Local Cache Replay Overhead	0%
Occupancy	
Achieved	91.3%
Theoretical	100%

Latency Hiding



- Instruction latencies:
  - Roughly 10-20 cycles for arithmetic operations
  - DRAM accesses have higher latencies (400-800 cycles)
- Instruction Level Parallelism (ILP)
  - Independent instructions between two dependent ones
  - ILP depends on the code, done by the compiler
- Switching to a different warp
  - If a warp must stall for *N* cycles due to dependencies, having *N* other warps with eligible instructions keeps the SM going
  - Switching among concurrently resident warps has no overhead
    - State (registers, shared memory) is partitioned, not stored/restored

Occupancy

- **Occupancy**: number of concurrent warps per SM, expressed as:
  - Absolute number of warps of threads that fit concurrently (e.g., 1..64), or
  - Ratio of warps that fit concurrently to architectural maximum (0..100%)

- Number of warps that fit determined by resource availability:
  - Threads per thread block
  - Registers per thread
  - Shared memory per thread block

Kepler SM resources:

- 64K 32-bit registers
- Up to 48 KB of shared memory
- Up to 2048 concurrent threads
- Up to 16 concurrent thread blocks

Occupancy and Performance

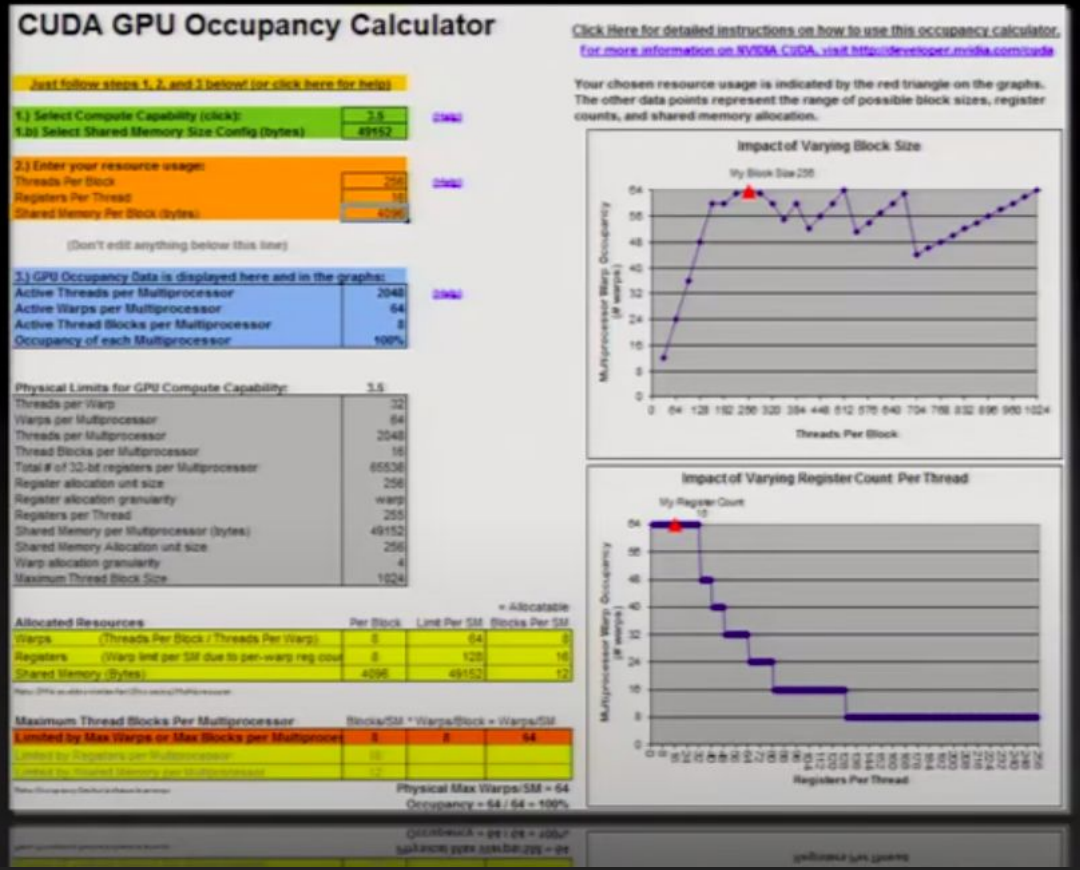
- Note that 100% occupancy isn't needed to reach maximum performance
  - Once the "needed" occupancy (enough warps to switch among to cover latencies) is reached, further increases won't improve performance
- Level of occupancy needed depends on the code
  - More independent work per thread -> less occupancy is needed
  - Memory-bound codes tend to need more occupancy
    - Higher latency than for arithmetic, need more work to hide it

Thread Block Size and Occupancy

- Thread block size is a multiple of warp size (32)
  - Even if you request fewer threads, hardware rounds up
- Thread blocks can be too small
  - Kepler SM can run up to 16 thread blocks concurrently
  - SM can reach the block count limit before reaching good occupancy
    - E.g.: 1-warp blocks = 16 warps/SM on Kepler (25% occ – probably not enough)
- Thread blocks can be too big
  - Enough SM resources for more threads, but not enough for a whole block
  - A thread block isn't started until resources are available for all of its threads

CUDA Occupancy Calculator

- Analyze effect of resource consumption on occupancy



Optimizing Memory Throughput

- Goal: utilize all available memory bandwidth
- Little's Law:

  # bytes in flight = latency * bandwidth

⇒ Increase parallelism (bytes in flight)

(or)

⇒ Reduce latency (time between requests)

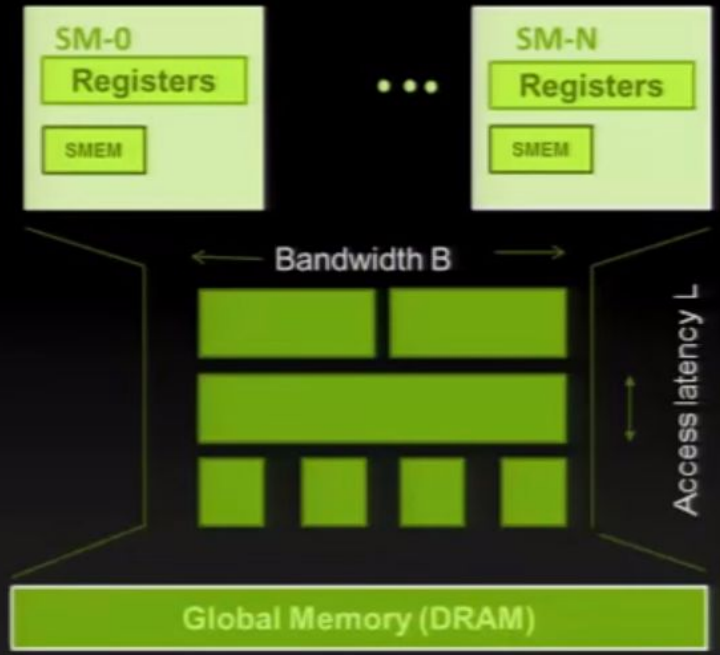
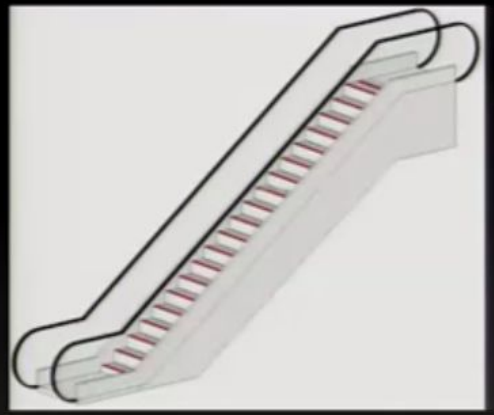


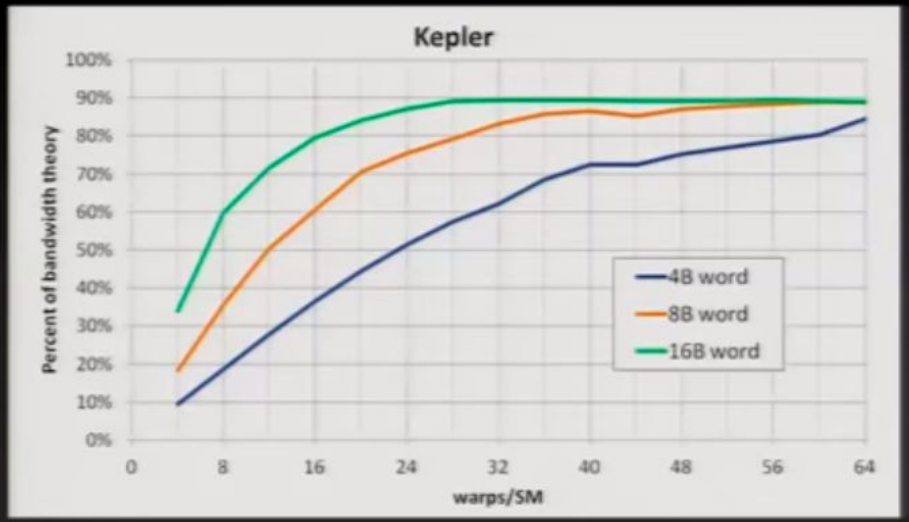
Illustration: Little's Law for Escalators

- Say the parameters of our escalator are:
  - 1 person fits on each step
  - Step arrives every 2 secs (*bandwidth*=0.5 persons/s)
  - 20 steps tall (*latency*=40 seconds)
- 1 person in flight: 0.025 persons/s achieved
- To saturate bandwidth:
  - Need 1 person arriving every 2 s
  - Means we'll need 20 persons in flight
- The idea: Bandwidth × Latency
  - It takes *latency* time units for the first person to arrive
  - We need *bandwidth* persons to get on the escalator every time unit



Requests per Thread and Performance

- Experiment: vary size of accesses by threads of a warp, check performance
  - Memcopy kernel: each warp has 2 concurrent requests (one write and the read following it)



Accesses by a warp:

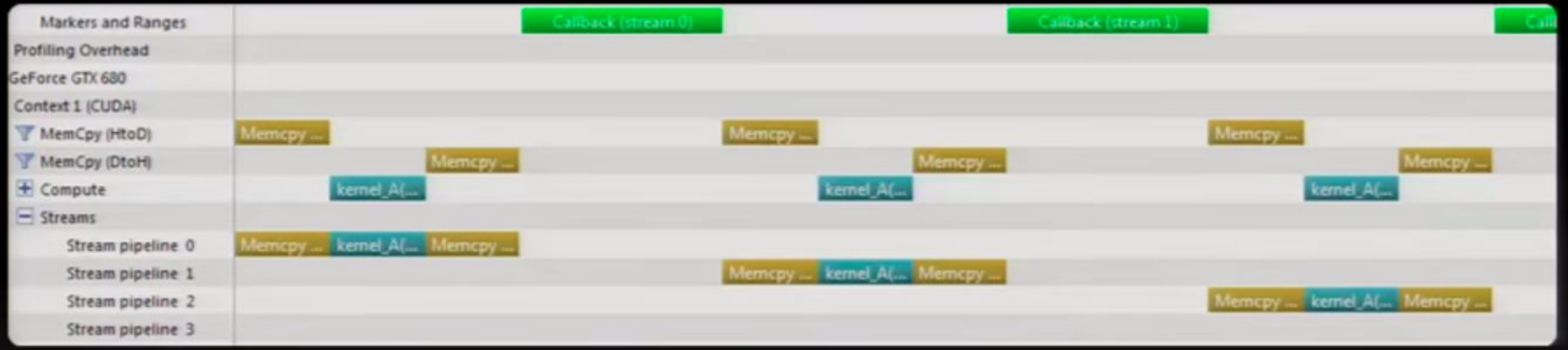
4B words: 1 line

8B words: 2 lines

16B words: 4 lines

To achieve same throughput at lower occupancy or with smaller words, need more independent requests per warp

Asynchronicity



- This is the kind of case we would be concerned about
  - Found the top kernel, but the GPU is mostly idle – *that* is our bottleneck
  - Need to overlap CPU/GPU computation and PCIe transfers

Parallelize: Achieve Asynchronicity

What we *want* to see is maximum overlap of all engines

GPU Optimization Fundamentals

- Find ways to parallelize sequential code
- Kernel optimizations
  - Launch configuration
  - Global memory throughput
  - Shared memory access
  - Instruction throughput / control flow
- Optimization of CPU-GPU interaction
  - Maximizing PCIe throughput
  - Overlapping kernel execution with memory copies