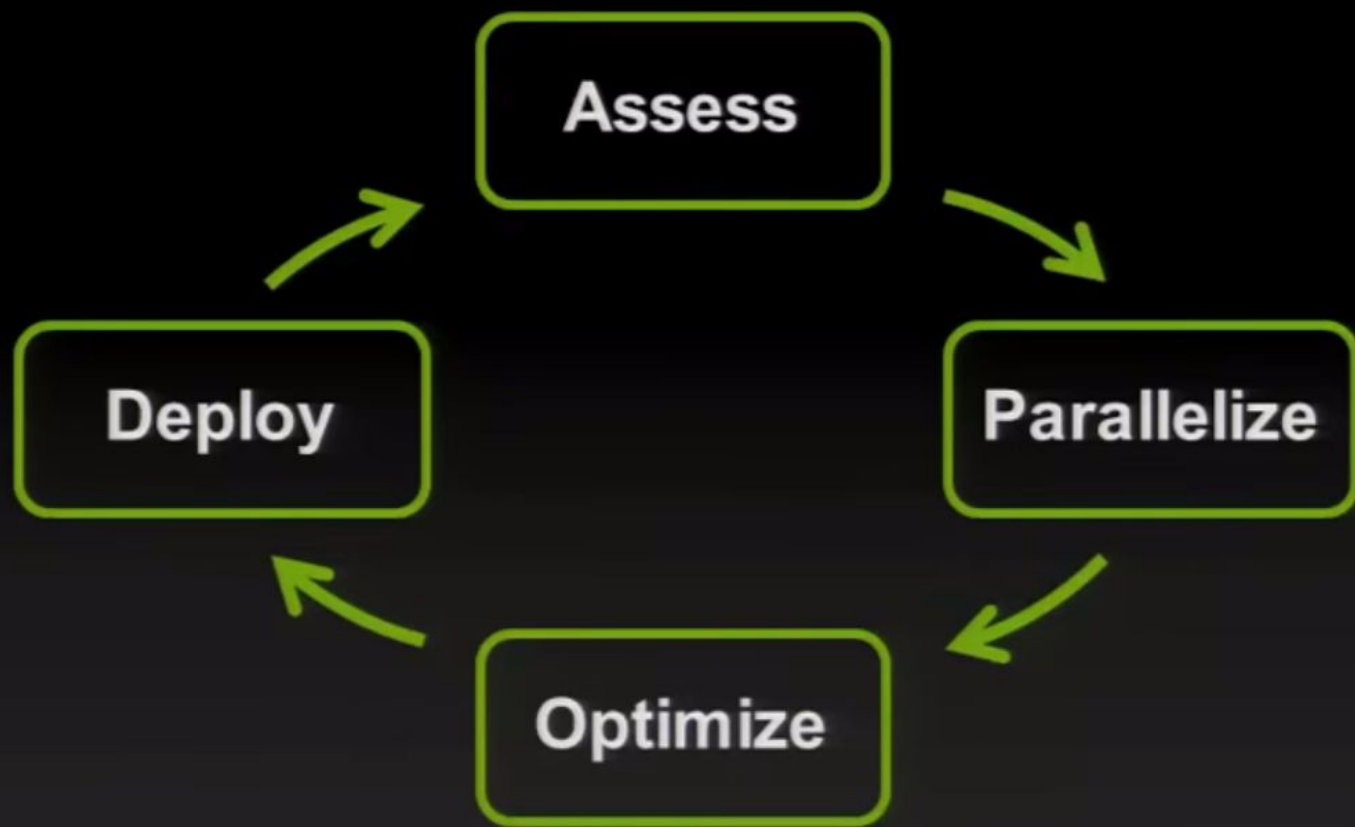


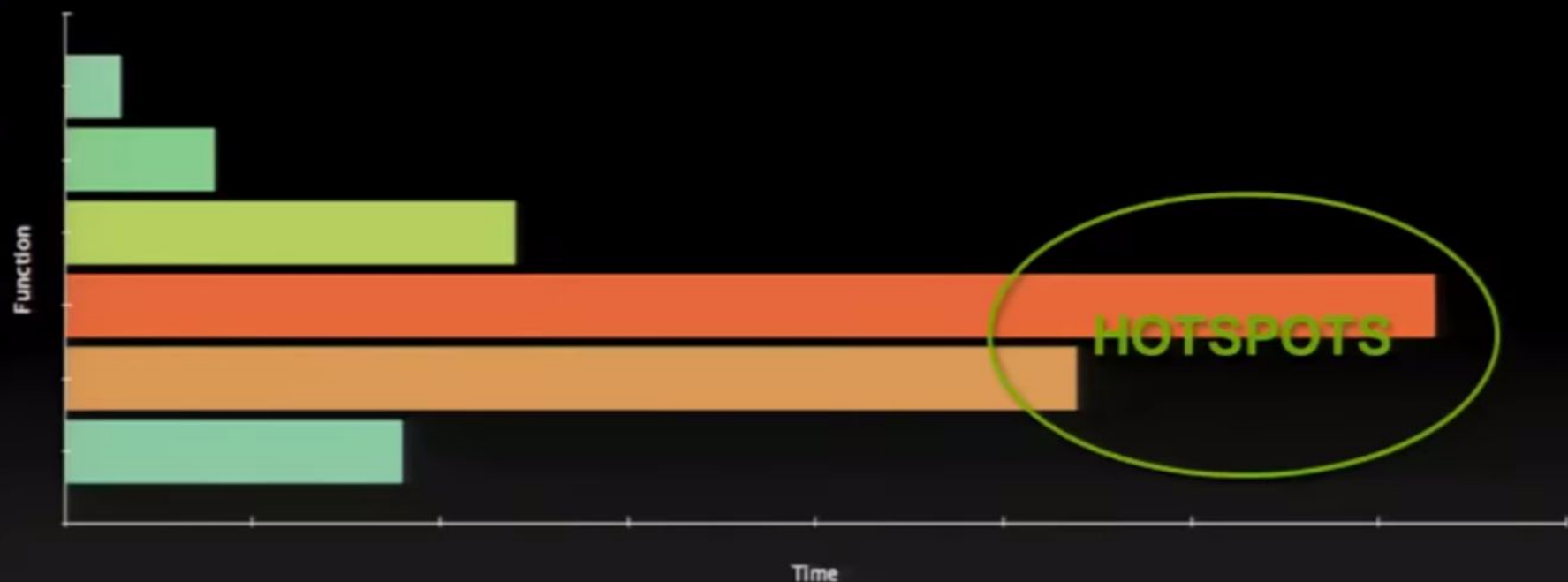
CUDA Optimizations - 1

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

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 axpbyoct_kernel<int=256>		5	5	28	0.35	4,442,227	157,857	158,651	159,394	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

Kernel Name	Processors	Time (s)	Speedup	Efficiency (%)
kernel_name_00000000	1	1000000	1.000000	100.000000
kernel_name_00000000	2	500000	2.000000	93.000000
kernel_name_00000000	4	250000	4.000000	93.000000
kernel_name_00000000	8	125000	8.000000	93.000000
kernel_name_00000000	16	62500	16.000000	93.000000
kernel_name_00000000	32	31250	32.000000	93.000000
kernel_name_00000000	64	15625	64.000000	93.000000
kernel_name_00000000	128	7812.5	128.000000	93.000000
kernel_name_00000000	256	3906.25	256.000000	93.000000
kernel_name_00000000	512	1953.125	512.000000	93.000000
kernel_name_00000000	1024	976.5625	1024.000000	93.000000
kernel_name_00000000	2048	488.28125	2048.000000	93.000000
kernel_name_00000000	4096	244.140625	4096.000000	93.000000
kernel_name_00000000	8192	122.0703125	8192.000000	93.000000
kernel_name_00000000	16384	61.03515625	16384.000000	93.000000
kernel_name_00000000	32768	30.517578125	32768.000000	93.000000
kernel_name_00000000	65536	15.2587890625	65536.000000	93.000000
kernel_name_00000000	131072	7.62939453125	131072.000000	93.000000
kernel_name_00000000	262144	3.814697265625	262144.000000	93.000000
kernel_name_00000000	524288	1.9073486328125	524288.000000	93.000000
kernel_name_00000000	1048576	0.95367431640625	1048576.000000	93.000000
kernel_name_00000000	2097152	0.476837158203125	2097152.000000	93.000000
kernel_name_00000000	4194304	0.2384185791015625	4194304.000000	93.000000
kernel_name_00000000	8388608	0.11920928955078125	8388608.000000	93.000000
kernel_name_00000000	16777216	0.059604644775390625	16777216.000000	93.000000
kernel_name_00000000	33554432	0.0298023223876953125	33554432.000000	93.000000
kernel_name_00000000	67108864	0.01490116119384765625	67108864.000000	93.000000
kernel_name_00000000	134217728	0.007450580596923828125	134217728.000000	93.000000
kernel_name_00000000	268435456	0.0037252902984619140625	268435456.000000	93.000000
kernel_name_00000000	536870912	0.00186264514923095703125	536870912.000000	93.000000
kernel_name_00000000	1073741824	0.000931322574615478515625	1073741824.000000	93.000000
kernel_name_00000000	2147483648	0.0004656612873077392578125	2147483648.000000	93.000000
kernel_name_00000000	4294967296	0.00023283064365386962890625	4294967296.000000	93.000000
kernel_name_00000000	8589934592	0.000116415321826934814453125	8589934592.000000	93.000000
kernel_name_00000000	17179869184	5.820766091289306640625e-05	17179869184.000000	93.000000
kernel_name_00000000	34359738368	2.9103830456446533203125e-05	34359738368.000000	93.000000
kernel_name_00000000	68719476736	1.45519152282232666015625e-05	68719476736.000000	93.000000
kernel_name_00000000	137438953472	7.27595761411163330078125e-06	137438953472.000000	93.000000
kernel_name_00000000	274877906944	3.637978807055816650390625e-06	274877906944.000000	93.000000
kernel_name_00000000	549755813888	1.8189894035279083251953125e-06	549755813888.000000	93.000000
kernel_name_00000000	1099511627776	9.0949470176395416259765625e-07	1099511627776.000000	93.000000
kernel_name_00000000	2199023255552	4.54747350881977081298828125e-07	2199023255552.000000	93.000000
kernel_name_00000000	4398046511104	2.273736754409885406494140625e-07	4398046511104.000000	93.000000
kernel_name_00000000	8796093022208	1.1368683772049427032470703125e-07	8796093022208.000000	93.000000
kernel_name_00000000	17592186044416	5.684341886024713516235390625e-08	17592186044416.000000	93.000000
kernel_name_00000000	35184372088832	2.8421709430123567581176953125e-08	35184372088832.000000	93.000000
kernel_name_00000000	70368744177664	1.42108547150617837905884765625e-08	70368744177664.000000	93.000000
kernel_name_00000000	140737488355328	7.10542735753089189529423828125e-09	140737488355328.000000	93.000000
kernel_name_00000000	281474976710656	3.552713678765445947647119140625e-09	281474976710656.000000	93.000000
kernel_name_00000000	562949953421312	1.7763568393827229738235595703125e-09	562949953421312.000000	93.000000
kernel_name_00000000	1125899906842624	8.8817841969136148691177978515625e-10	1125899906842624.000000	93.000000
kernel_name_00000000	2251799813685248	4.44089209845680743455889892578125e-10	2251799813685248.000000	93.000000
kernel_name_00000000	4503599627370496	2.220446049228403717279449462890625e-10	4503599627370496.000000	93.000000
kernel_name_00000000	9007199254740992	1.1102230246142018586397247314453125e-10	9007199254740992.000000	93.000000
kernel_name_00000000	18014398509481984	5.5511151230710092931988236571875e-11	18014398509481984.000000	93.000000
kernel_name_00000000	36028797018963968	2.77555756153550464659941182859375e-11	36028797018963968.000000	93.000000
kernel_name_00000000	72057594037927936	1.387778780767752323299705914296875e-11	72057594037927936.000000	93.000000
kernel_name_00000000	144115188075855872	6.9388939038387616164985295721484375e-12	144115188075855872.000000	93.000000
kernel_name_00000000	288230376151711744	3.46944695191938080824926478607421875e-12	288230376151711744.000000	93.000000
kernel_name_00000000	576460752303423488	1.734723475959690404124932393037109375e-12	576460752303423488.000000	93.000000
kernel_name_00000000	1152921504606846976	8.67361737979845202062466196518546875e-13	1152921504606846976.000000	93.000000
kernel_name_00000000	2305843009213693952	4.336808689899226010312330982592734375e-13	2305843009213693952.000000	93.000000
kernel_name_00000000	4611686018427387904	2.1684043449496130051561654912963671875e-13	4611686018427387904.000000	93.000000
kernel_name_00000000	9223372036854775808	1.08420217247480650257808274564818359375e-13	9223372036854775808.000000	93.000000
kernel_name_00000000	18446744073709551616	5.42101086237403251289041372824091796875e-14	18446744073709551616.000000	93.000000
kernel_name_00000000	36893488147419103232	2.710505431187016256445206864120458984375e-14	36893488147419103232.000000	93.000000
kernel_name_00000000	73786976294838206464	1.3552527155935081282226034320602294921875e-14	73786976294838206464.000000	93.000000
kernel_name_00000000	147573952589676412928	6.77626357796754064111301716030114724609375e-15	147573952589676412928.000000	93.000000
kernel_name_00000000	295147905179352825856	3.388131788983770320556508580150573623046875e-15	295147905179352825856.000000	93.000000
kernel_name_00000000	590295810358705651712	1.6940658944918851602782542900752868115234375e-15	590295810358705651712.000000	93.000000
kernel_name_00000000	1180591620717411303424	8.4703294724594258013912714500764344076171875e-16	1180591620717411303424.000000	93.000000
kernel_name_00000000	2361183241434822606848	4.2351647362297129006956357250382172038089375e-16	2361183241434822606848.000000	93.000000
kernel_name_00000000	4722366482869645213696	2.11758236811485645034781786251910860190446875e-16	4722366482869645213696.000000	93.000000
kernel_name_00000000	9444732965739290427392	1.058791184057428225173908931259554300952234375e-16	9444732965739290427392.000000	93.000000
kernel_name_00000000	18889465931478580854784	5.293955920287141125869544656259771504761171875e-17	18889465931478580854784.000000	93.000000
kernel_name_00000000	37778931862957161709568	2.6469779601435705629347723281298857523805859375e-17	37778931862957161709568.000000	93.000000
kernel_name_00000000	75557863725914323419136	1.32348898007178528146738616406494287619029296875e-17	75557863725914323419136.000000	93.000000
kernel_name_00000000	151115727451828646838272	6.61744490035892640733693082032471438095146484375e-18	151115727451828646838272.000000	93.000000
kernel_name_00000000	302231454903657293676544	3.308722450179463203668465410162357190475732241875e-18	302231454903657293676544.000000	93.000000
kernel_name_00000000	604462909807314587353088	1.6543612250897316018342327050811785952378661209375e-18	604462909807314587353088.000000	93.000000
kernel_name_00000000	1208925819614629174706176	8.2718061254486580091711635254058929761893306046875e-19	1208925819614629174706176.000000	93.000000
kernel_name_00000000	2417851639229258349412352	4.13590306272432900458558176270294648809466530234375e-19	2417851639229258349412352.000000	93.000000
kernel_name_00000000	4835703278458516698824704	2.067951531362164502292790881351473244047332651171875e-19	4835703278458516698824704.000000	93.000000
kernel_name_00000000	9671406556917033397649408	1.0339757656810822511463954406757366220236663255859375e-19	9671406556917033397649408.000000	93.000000
kernel_name_00000000	19342813113834066795298816	5.16987882840541125573197720337868311011833316279296875e-20	19342813113834066795298816.000000	93.000000
kernel_name_00000000	38685626227668133590597632	2.584939414202705627865988601689341555059166576396484375e-20	38685626227668133590597632.000000	93.000000
kernel_name_00000000	77371252455336267181195264	1.2924697071013528139329943008446707775295832881982421875e-20	77371252455336267181195264.000000	93.000000
kernel_name_00000000	154742504910672534362390528	6.4623485355067640696649715042233538876479164409912109375e-21	154742504910672534362390528.000000	93.000000
kernel_name_00000000	309485009821345068724781056	3.23117426775338203483248575211167694382395822049561046875e-21	309485009821345068724781056.000000	93.000000
kernel_name_00000000	618970019642690137449562112	1.6155871338766910174162428760558384719119791102478051953125e-21	618970019642690137449562112.000000	93.000000
kernel_name_00000000	1237940039285380274899124224	8.0779356693834550870812114302791923595598955512390259765625e-22	1237940039285380274899124224.000000	93.000000
kernel_name_00000000	2475880078570760549798248448	4.03896783469172754354060571513959617977994777561951298828125e-22	2475880078570760549798248448.000000	93.000000
kernel_name_00000000	4951760157141521099596496896	2.019483917345863771770302857569798089889973887809756494140625e-22	4951760157141521099596496896.000000	93.000000
kernel_name_00000000	9903520314283042199192993792	1.0097419586729318858851514287848990449449869439048782470703125e-22	9903520314283042199192993792.000000	93.000000
kernel_name_00000000	19807040628566084398385987584	5.04870979336465942942575714392449522472494497195439137109375e-23	19807040628566084398385987584.000000	93.000000
kernel_name_00000000	39614081257132168796771975168	2.524354896682329714712878571962247612362472485977195685546875e-23	39614081257132168796771975168.000000	93.000000
kernel_name_00000000	79228162514264337593543950336	1.2621774483411648573564392859811238061812362429885978277234375e-23	79228162514264337593543950336.000000	93.000000
kernel_name_00000000	158456325028528675187087900672	6.3108872417058242867821964299056190309061812362429885978277234375e-24	158456325028528675187087900672.000000	93.000000
kernel_name_00000000	316912650057057350374175801344	3.15544362085291214339109821495280951545309061812362429885978277234375e-24	316912650057057350374175801344.000000	93.000000

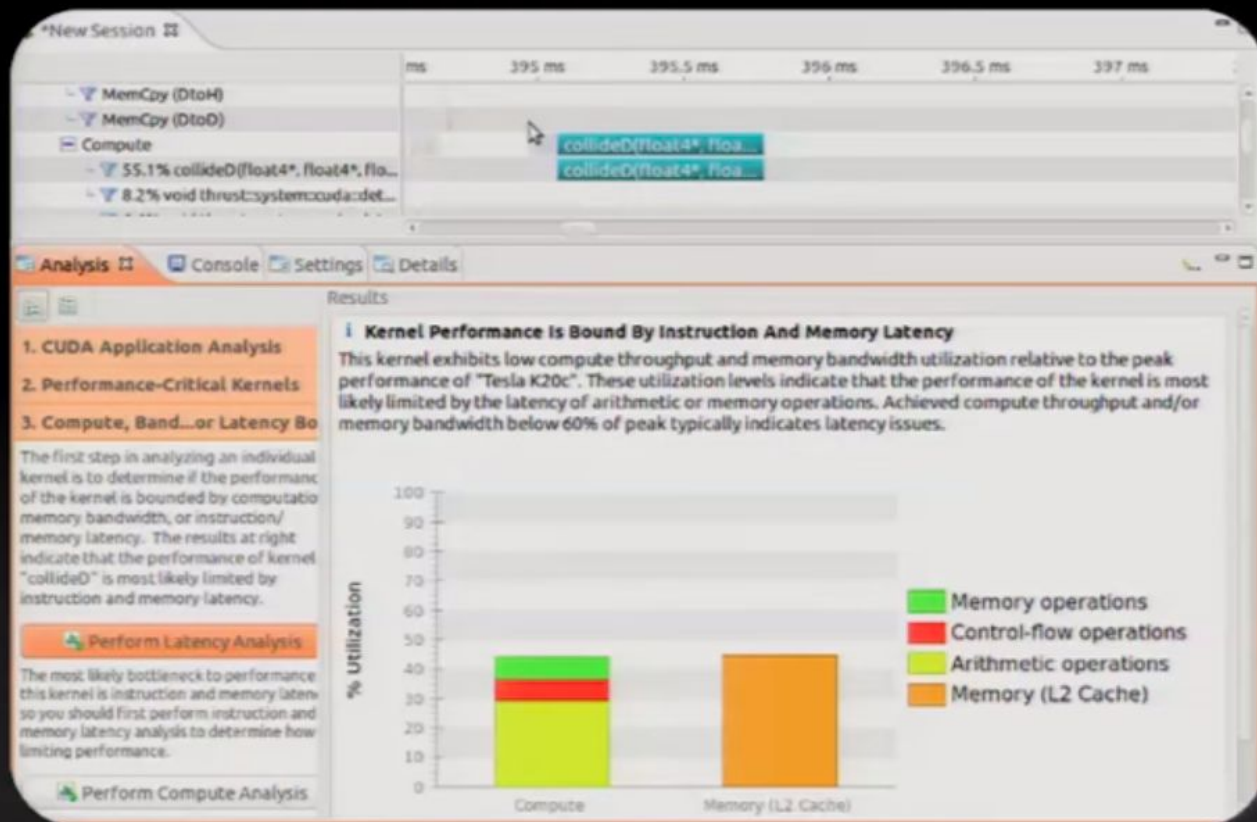
Assess: Speed of Light

- 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 $\frac{G\text{Bytes}/\text{sec}}{G\text{insns}/\text{sec}}$
 - 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:

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

Kernel	Op/s	Bandwidth (GB/s)	Peak Bandwidth (GB/s)	Efficiency (%)
SpMV	100000000	1.70000	2.00000	~93%
...

Parallelize

Applications

Libraries

Compiler
Directives

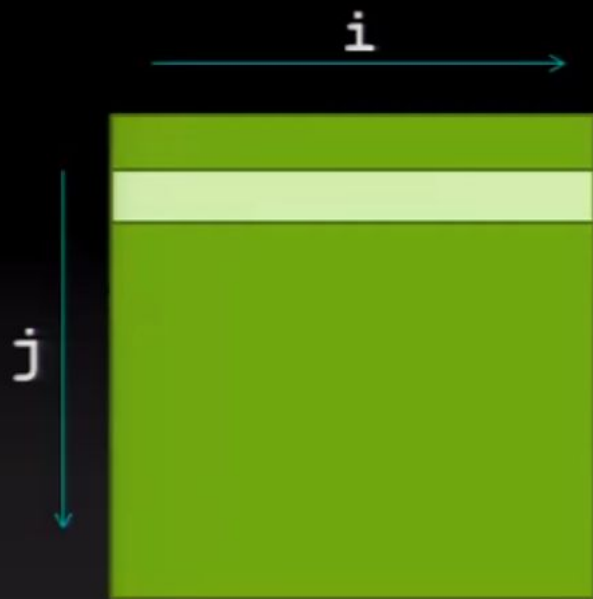
Programming
Languages

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);
```

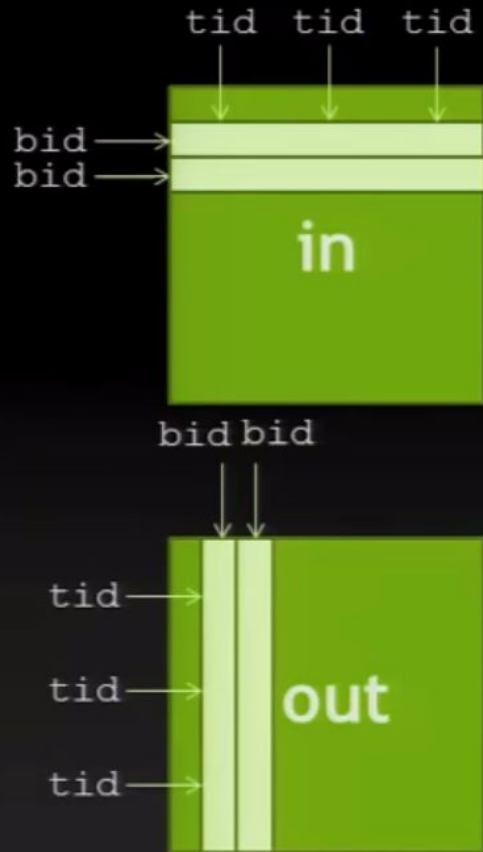
+ Quickly implemented

- 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)

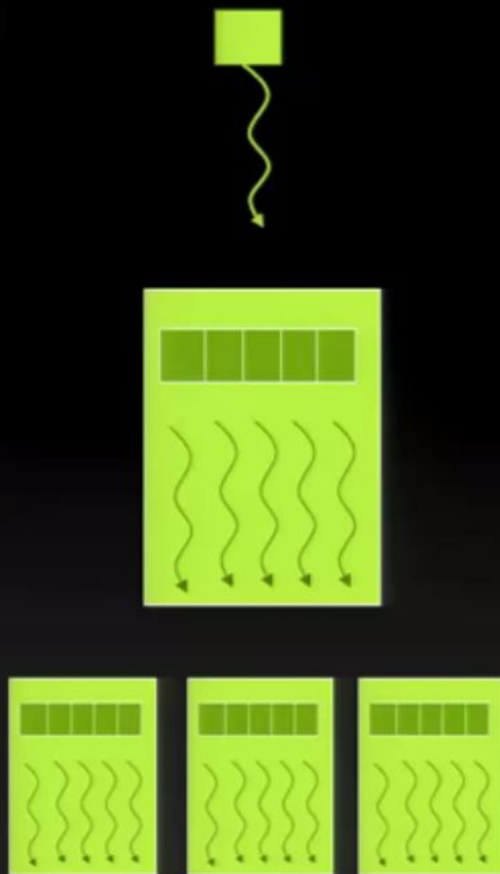
Per SMX:

- 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

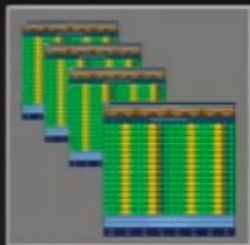
Hardware



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

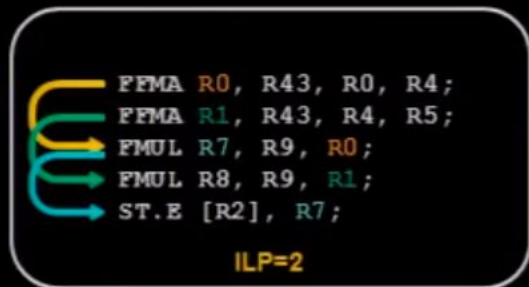
Occupancy

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

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

Start	588.755 ms
End	588.808 ms
Duration	53.344 μ s
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

CUDA GPU Occupancy Calculator

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

1.) Select Compute Capability (click): GPU

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

2.) Enter your resource usage:

Threads Per Block: GPU

Registers Per Thread:

Shared Memory Per Block (bytes):

(Don't edit anything below this line)

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

Active Threads per Multiprocessor: 2048 GPU

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
Warps per Multiprocessor	64
Threads per Multiprocessor	2048
Total 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 Blocks Per SM < Allocated

Warps (Threads Per Block / Threads Per Warp) 8 64 8

Registers (Warp limit per SM due to per-warp reg count) 8 128 16

Shared Memory (Bytes) 4096 49152 12

Maximum Thread Blocks Per Multiprocessor $\text{Blocks/SM} * \text{Warps/Block} = \text{Warps/SM}$

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

Limited by Registers per Multiprocessor 8

Limited by Shared Memory per Multiprocessor 12

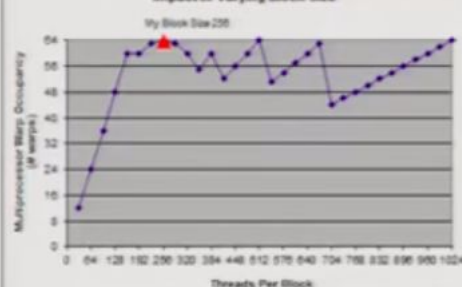
Physical Max Warps/SM = 64

Occupancy = $64 / 64 = 100\%$

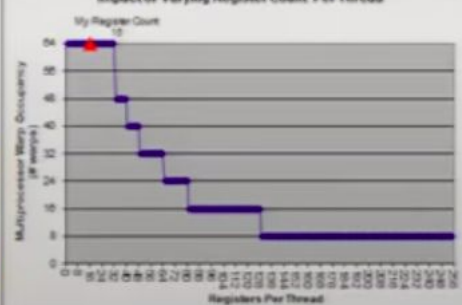
Click here for detailed instructions on how to use this occupancy calculator. For more information on NVIDIA CUDA, visit <http://developer.nvidia.com/cuda>

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.

Impact of Varying Block Size



Impact of Varying Register Count Per Thread



Occupancy = $64 / 64 = 100\%$

Occupancy = $64 / 64 = 100\%$

Occupancy = 100%

Optimizing Memory Throughput

- **Goal: utilize all available memory bandwidth**
 - **Little's Law:**
 $\# \text{ bytes in flight} = \text{latency} * \text{bandwidth}$
- ⇒ **Increase parallelism (bytes in flight)**
(or)
⇒ **Reduce latency (time between requests)**

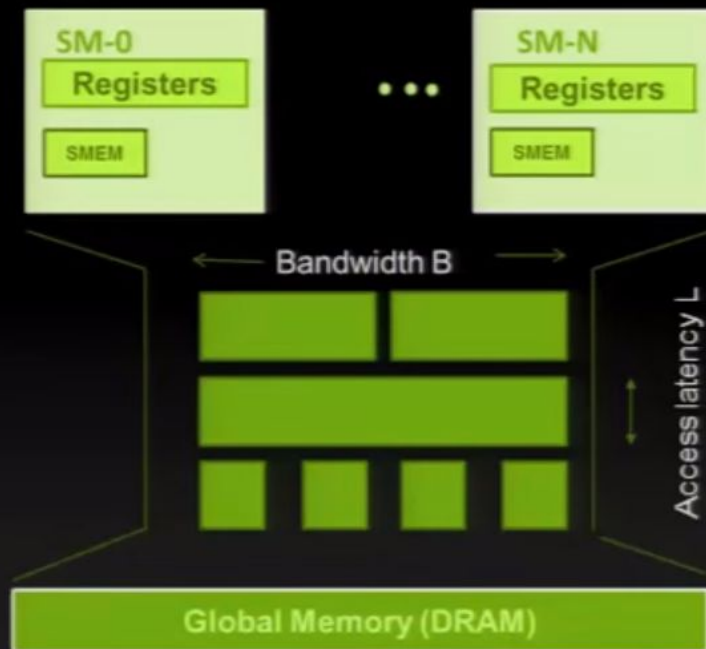
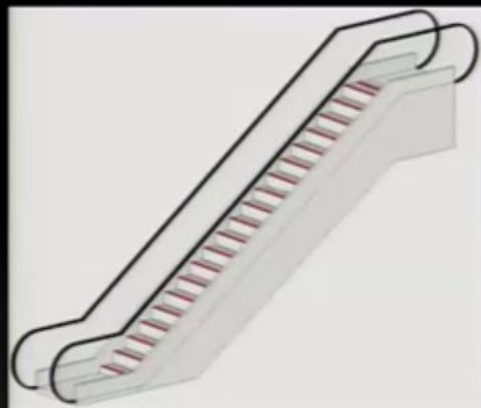


Illustration: Little's Law for Escalators

- 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

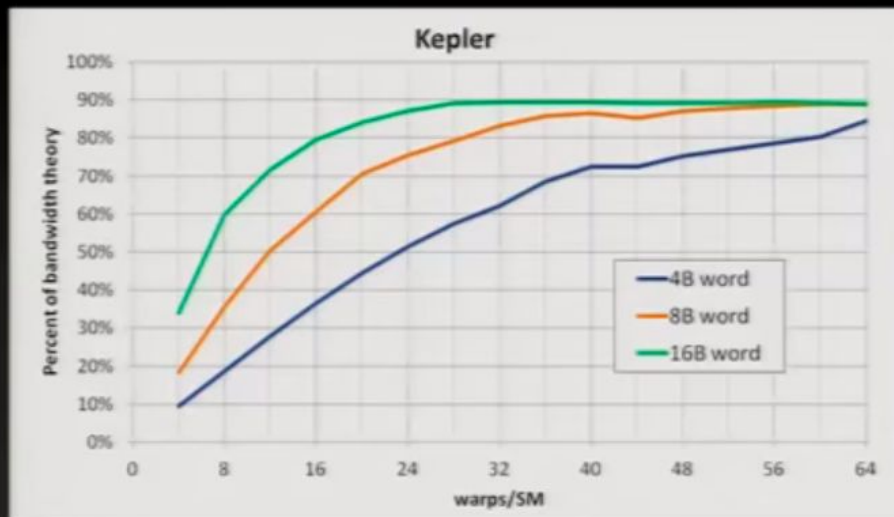
- **Experiment: vary size of accesses by threads of a warp, check performance**
 - Memcopy kernel: each warp has 2 concurrent requests (one write and the read following it)

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