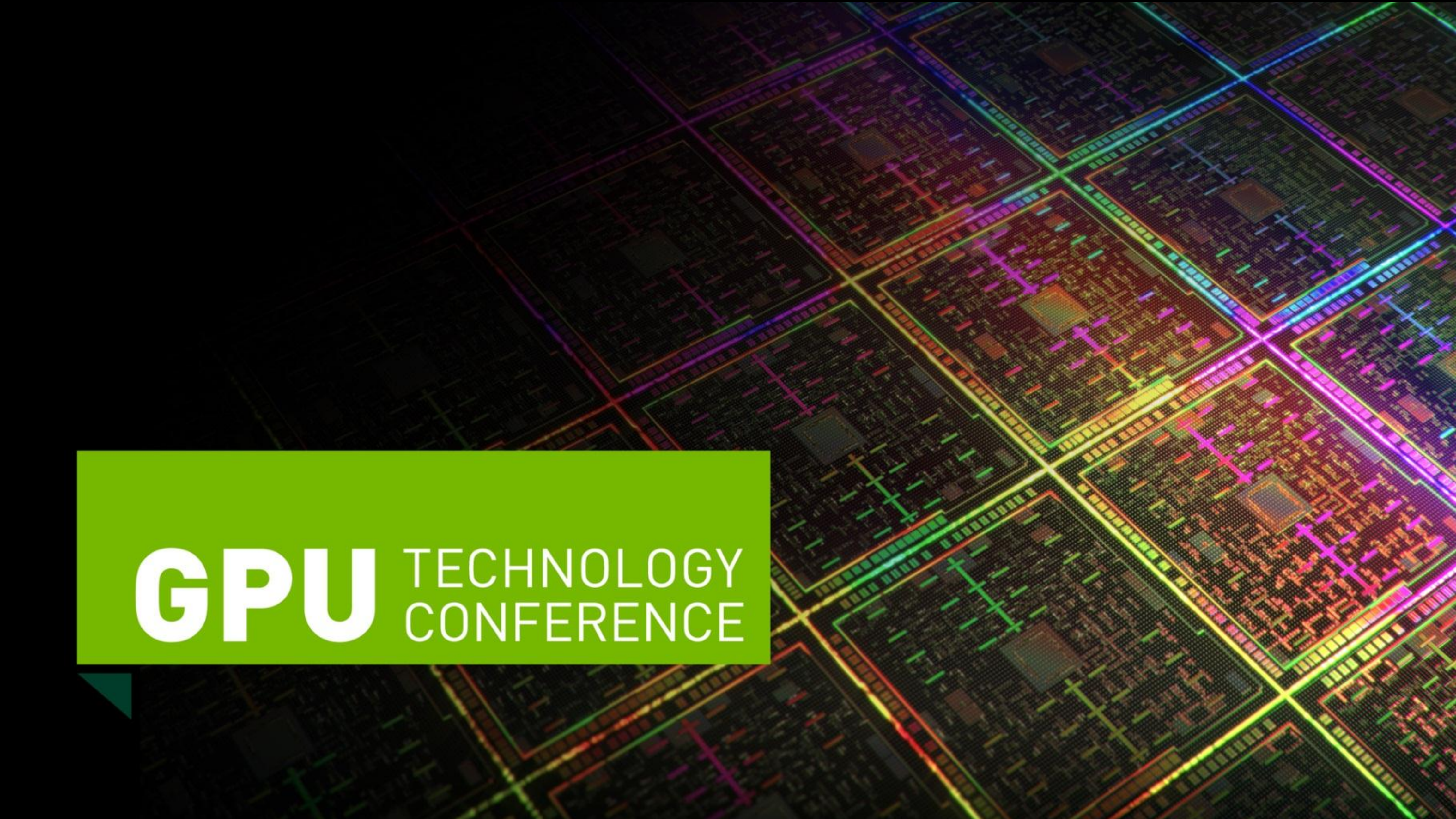


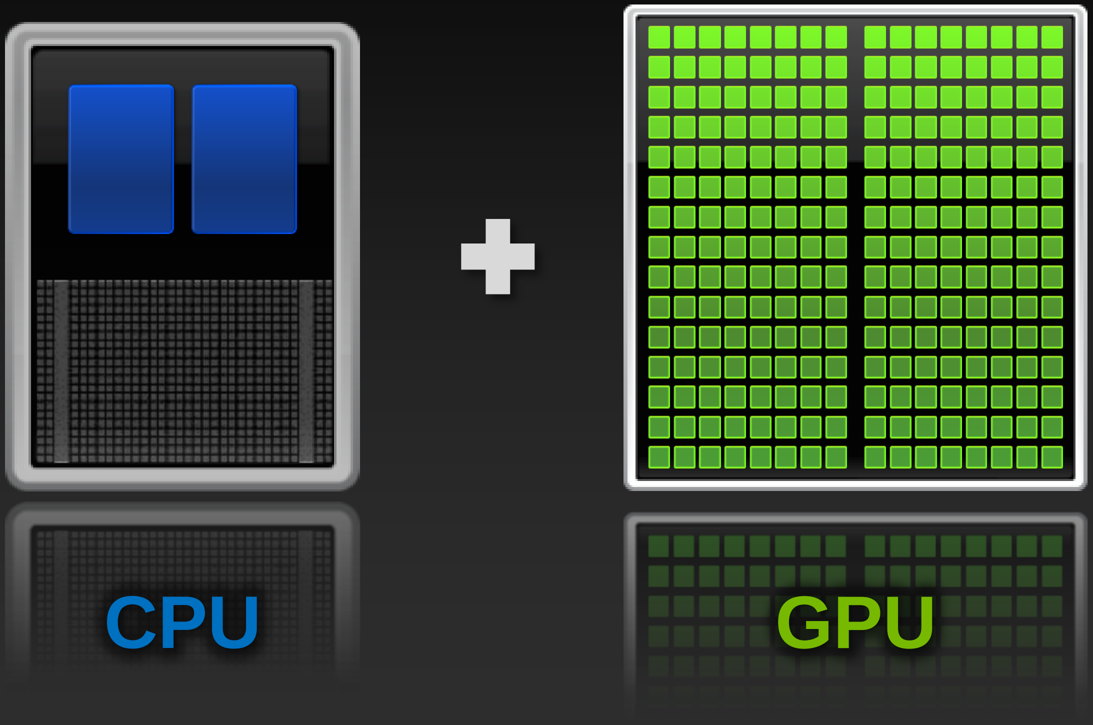
GPU TECHNOLOGY CONFERENCE

Profiling and Tuning OpenACC Code

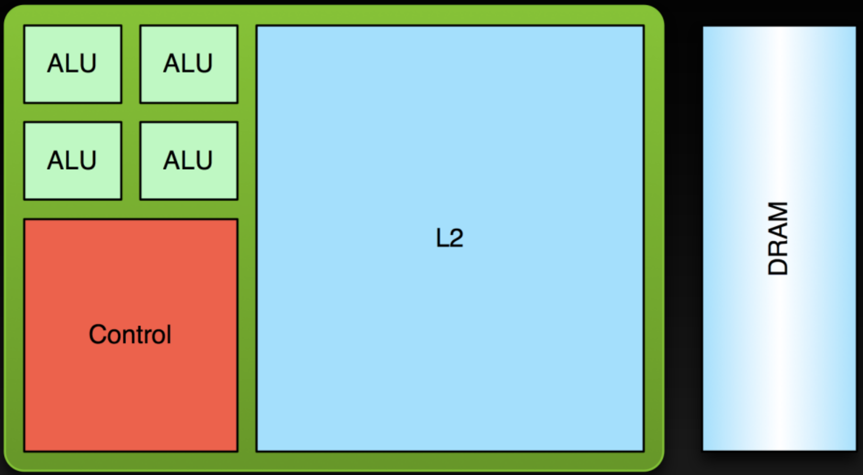
Cliff Woolley, NVIDIA
Developer Technology Group

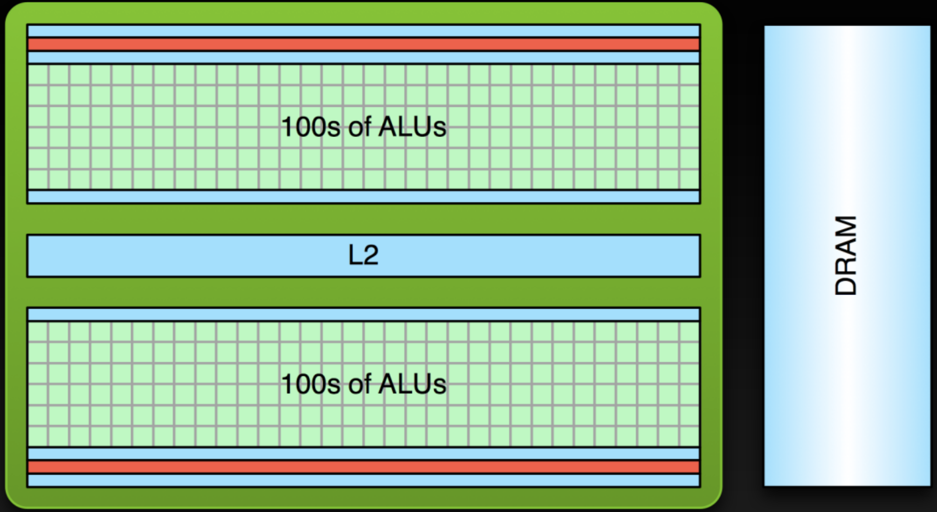
GPGPU Revolutionizes Computing

Latency Processor + Throughput processor



Low Latency or High Throughput?





CPU

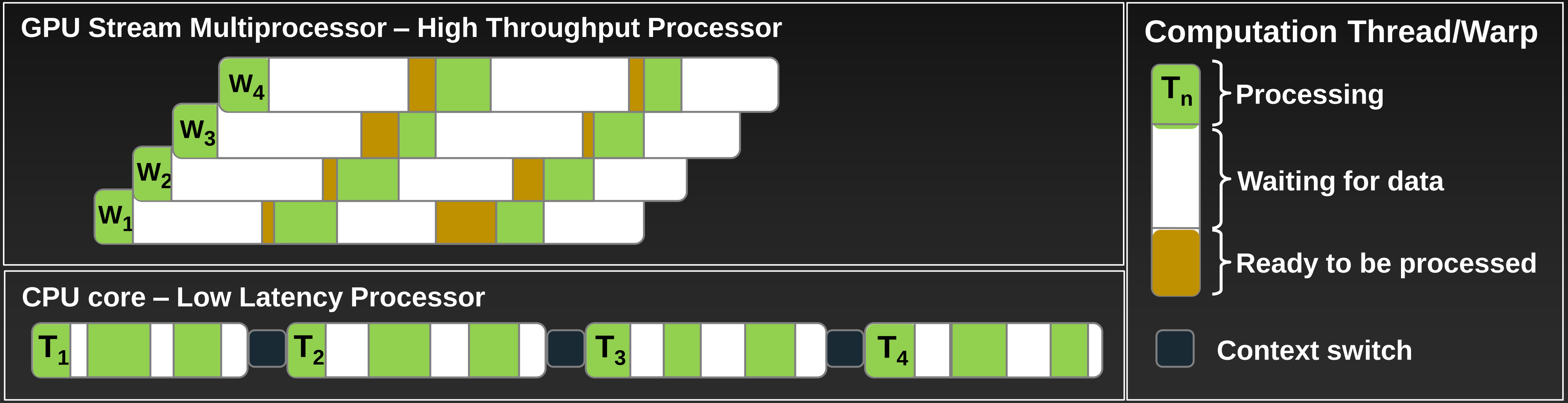
- Optimized for low-latency access to cached data sets
- Control logic for out-of-order and speculative execution

GPU

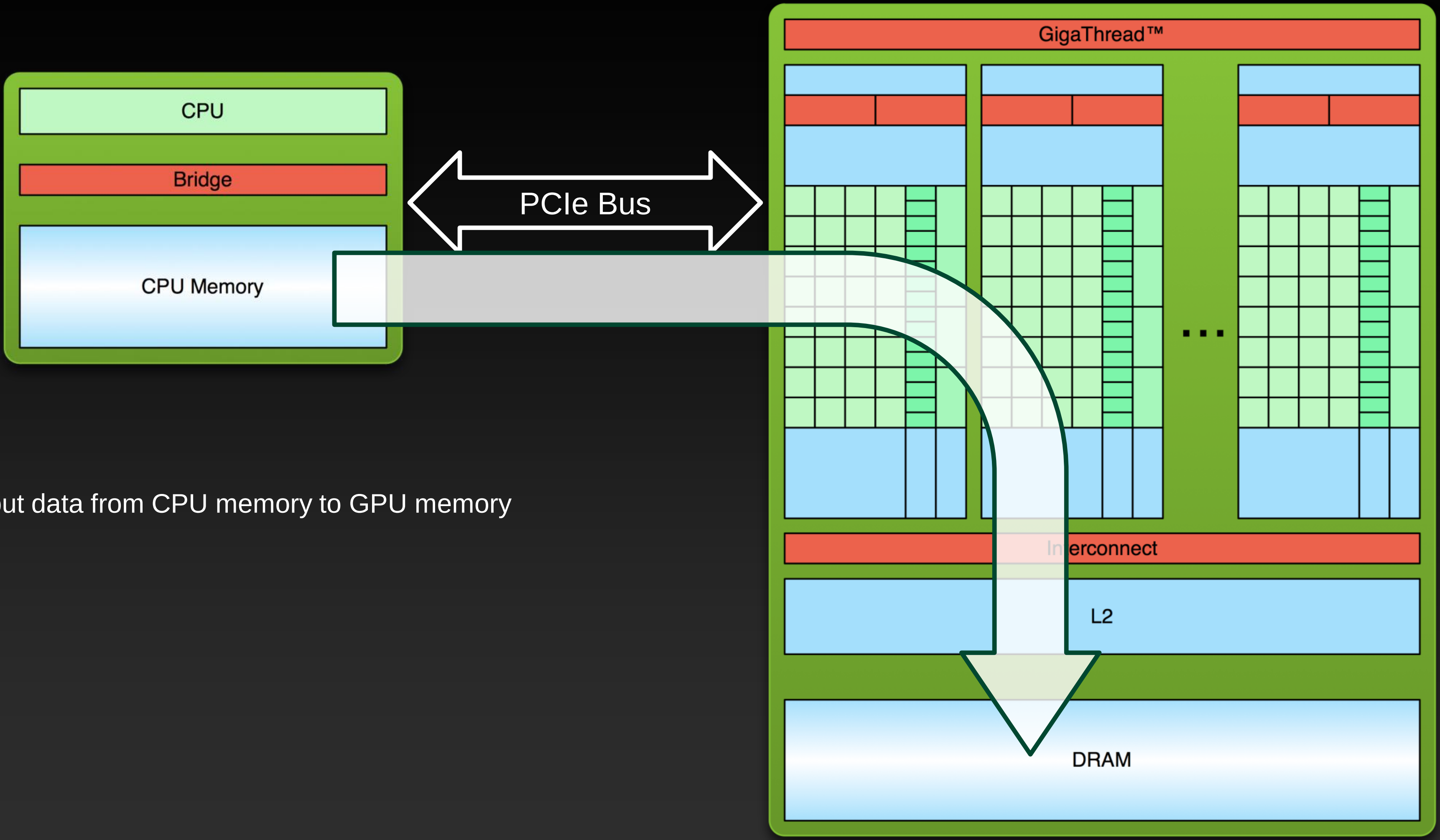
- Optimized for data-parallel, throughput computation
- Architecture tolerant of memory latency
- More transistors dedicated to computation

Low Latency or High Throughput?

- CPU architecture must minimize latency within each thread
- GPU architecture hides latency with computation from other thread warps

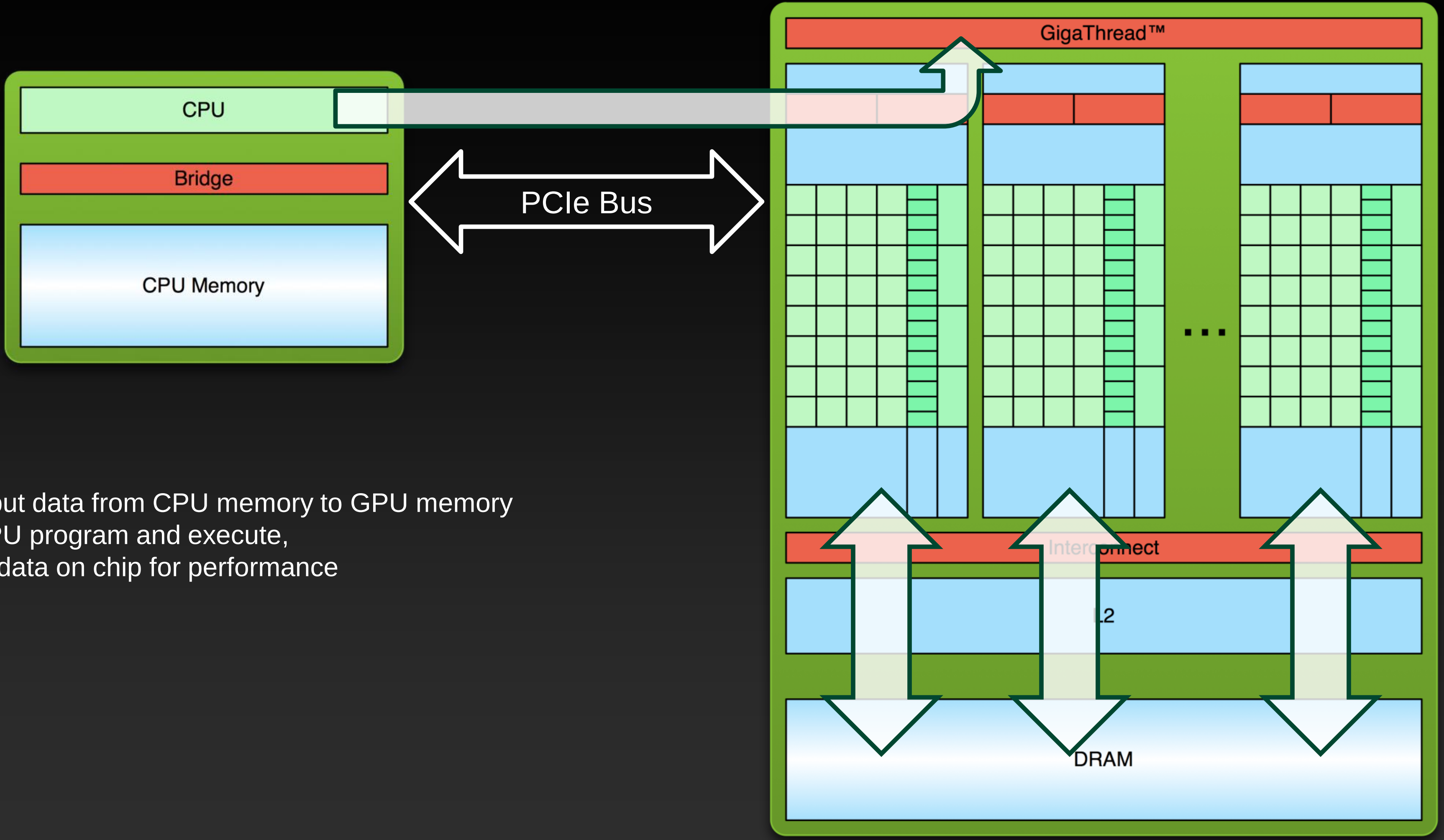


Processing Flow



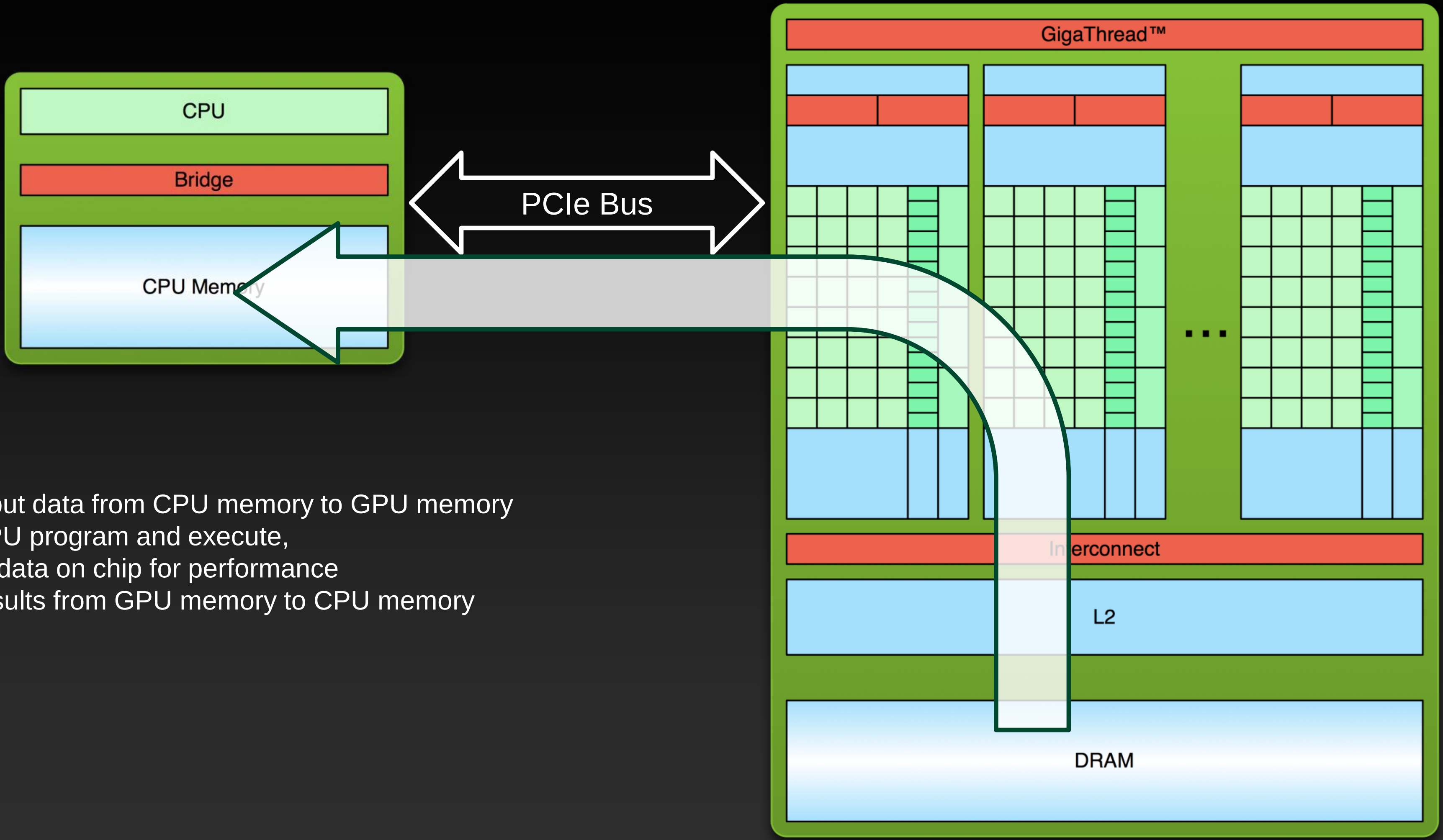
1. Copy input data from CPU memory to GPU memory

Processing Flow



1. Copy input data from CPU memory to GPU memory
2. Load GPU program and execute, caching data on chip for performance

Processing Flow



1. Copy input data from CPU memory to GPU memory
2. Load GPU program and execute,
   caching data on chip for performance
3. Copy results from GPU memory to CPU memory

OpenACC and CUDA

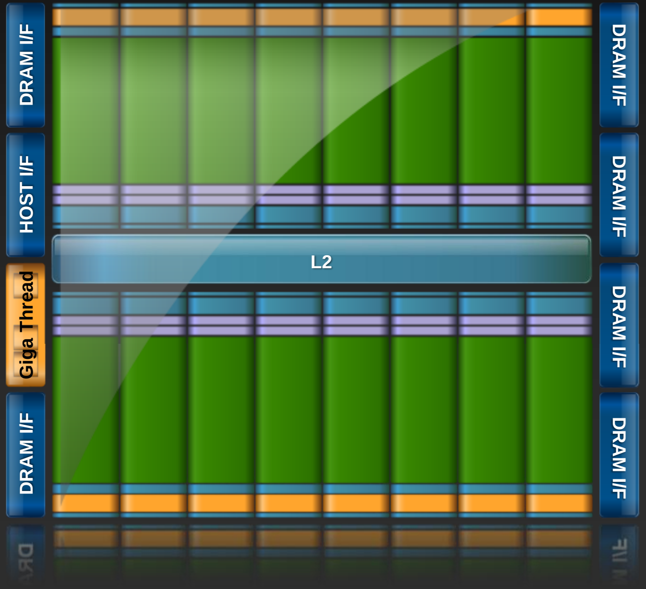
- OpenACC enables a compiler to target annotated C or Fortran code to accelerators such as NVIDIA CUDA-capable GPUs

- Note: CUDA refers to both a parallel computing platform and a parallel programming model

CUDA ARCHITECTURE REVIEW

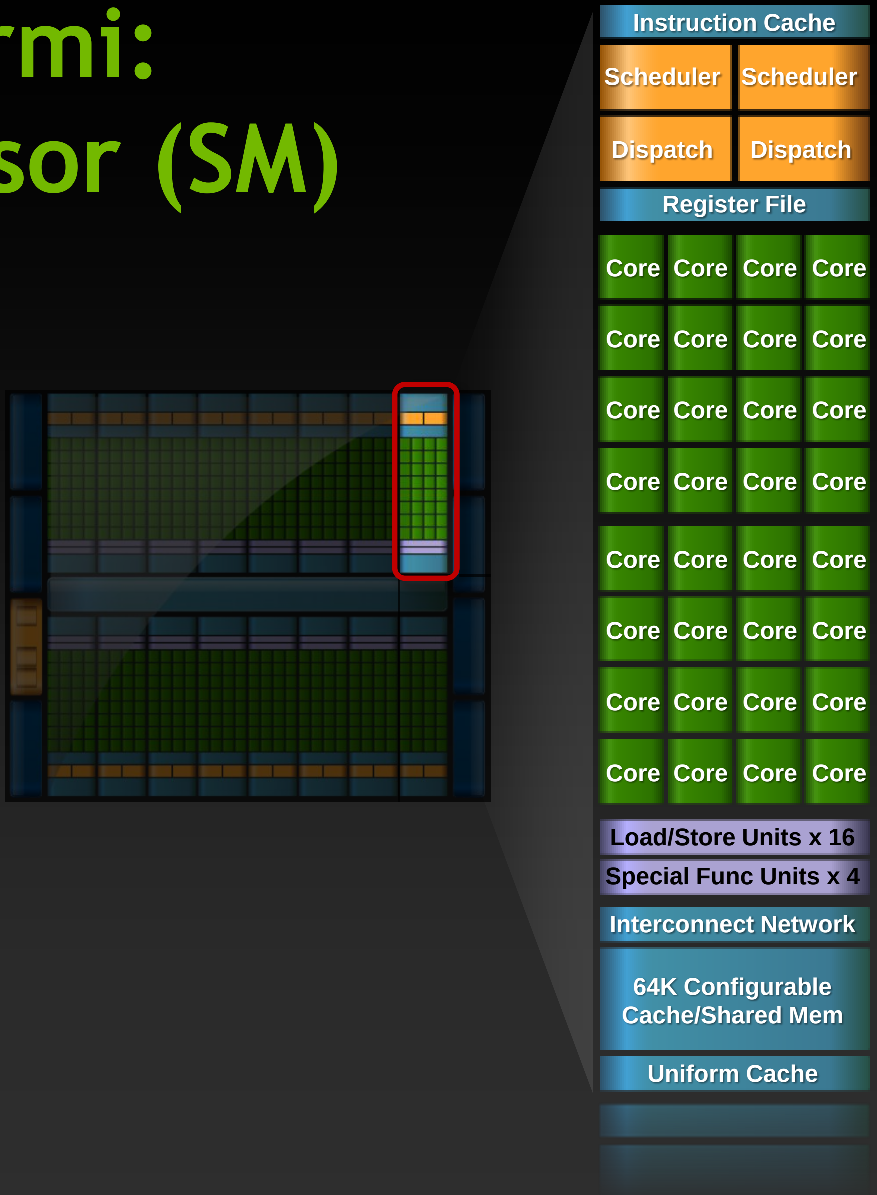
GPU Architecture: Two Main Components

- Global memory
  - Analogous to RAM in a CPU server
  - Accessible by both GPU and CPU
  - Currently up to 6 GB
  - Bandwidth currently up to almost 180 GB/s for Tesla products
  - ECC on/off option for Quadro and Tesla products
- Streaming Multiprocessors (SMs)
  - Perform the actual computations
  - Each SM has its own:
    - Control units, registers, execution pipelines, caches



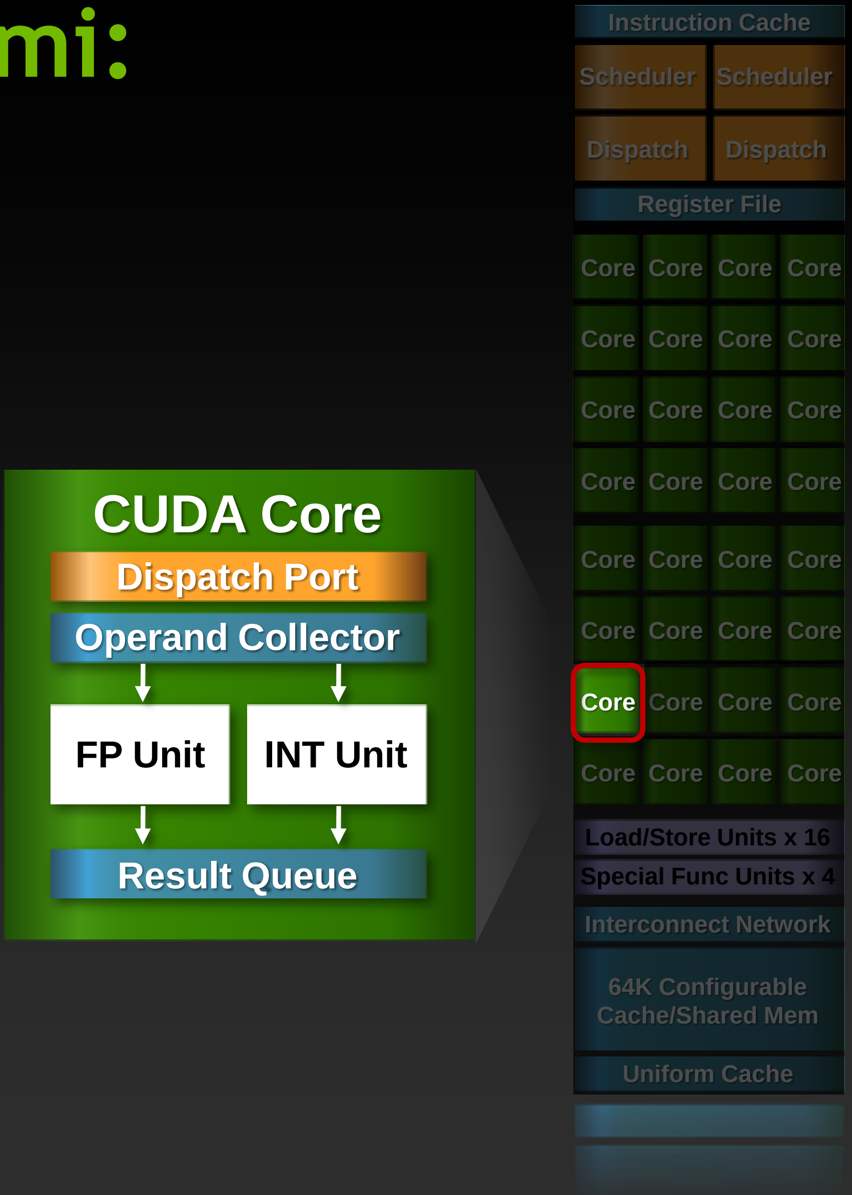
GPU Architecture - Fermi: Streaming Multiprocessor (SM)

- 32 CUDA Cores per SM
  - 32 fp32 ops/clock
  - 16 fp64 ops/clock
  - 32 int32 ops/clock
- 2 warp schedulers
  - Up to 1536 threads concurrently
- 4 special-function units
- 64KB shared mem + L1 cache
- 32K 32-bit registers



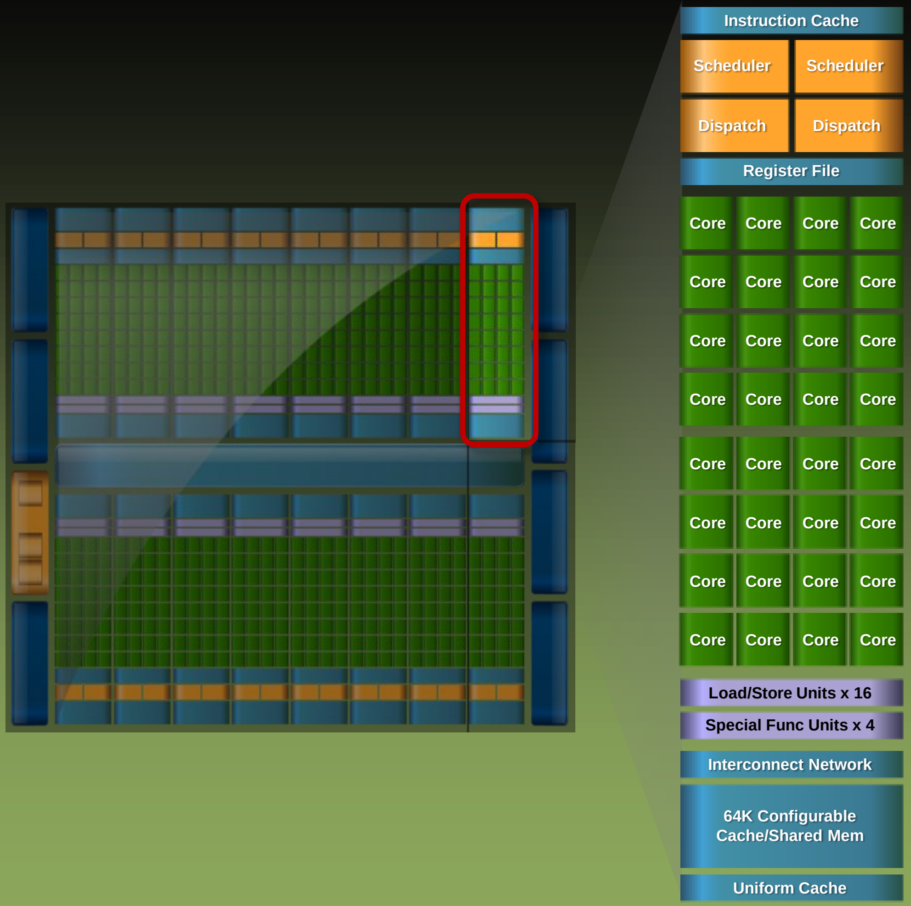
GPU Architecture - Fermi: CUDA Core

- Floating point & Integer unit
  - IEEE 754-2008 floating-point standard
  - Fused multiply-add (FMA) instruction for both single and double precision
- Logic unit
- Move, compare unit
- Branch unit

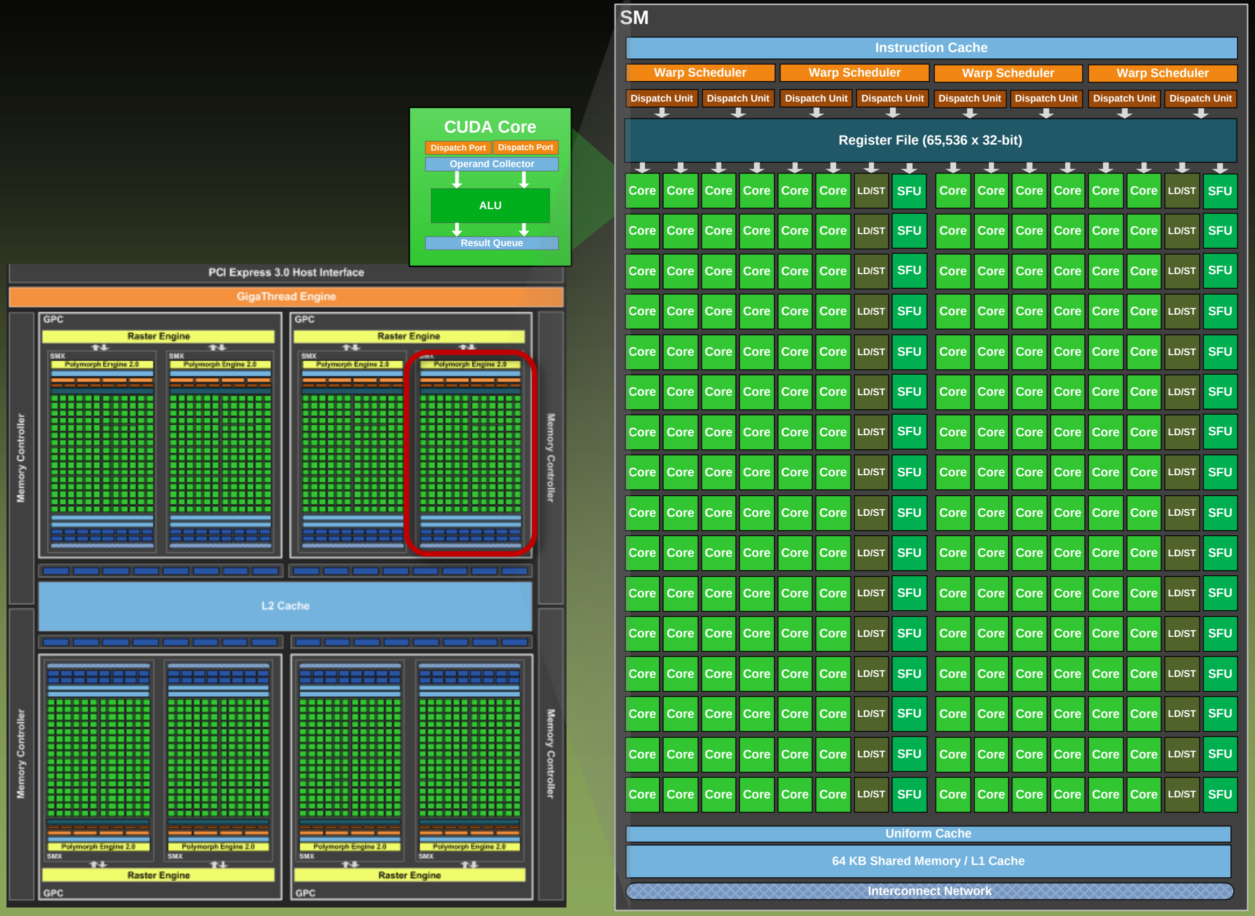


Kepler

Fermi



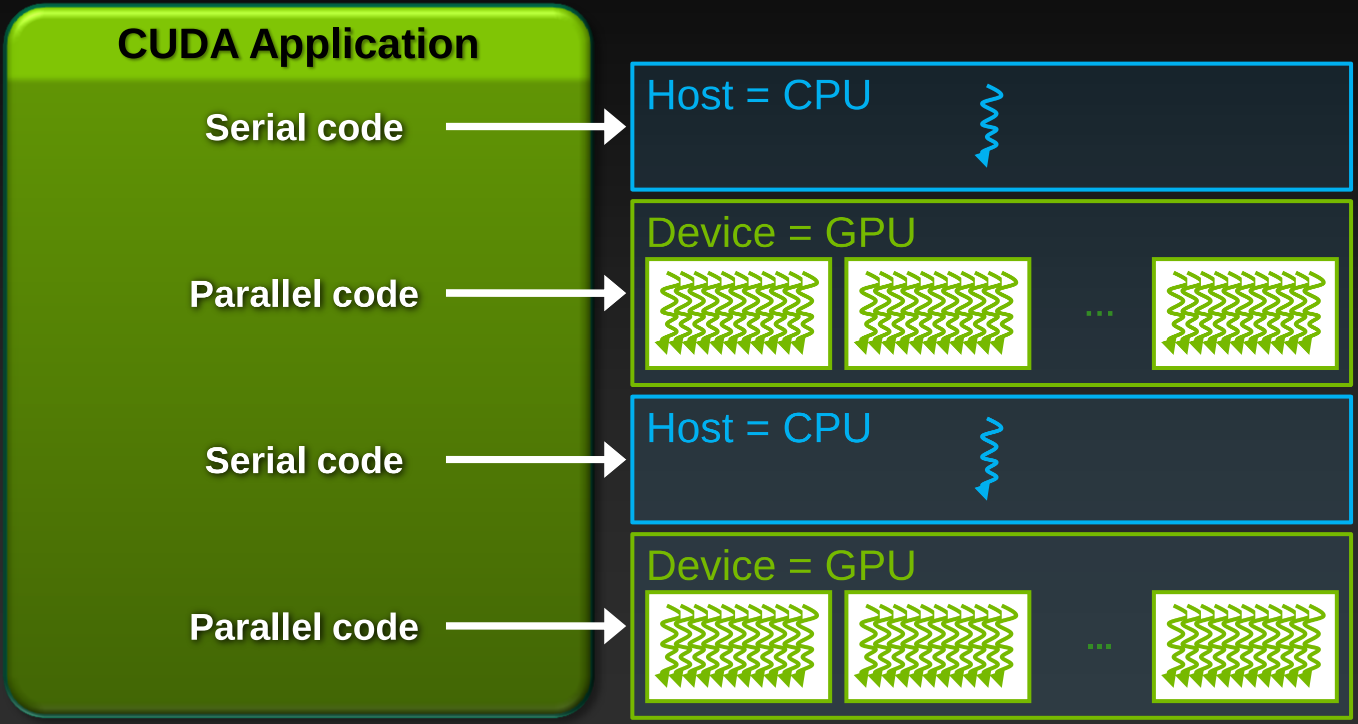
Kepler



CUDA PROGRAMMING MODEL REVIEW

Anatomy of a CUDA Application

- Serial code executes in a Host (CPU) thread
- Parallel code executes in many Device (GPU) threads across multiple processing elements



CUDA Kernels

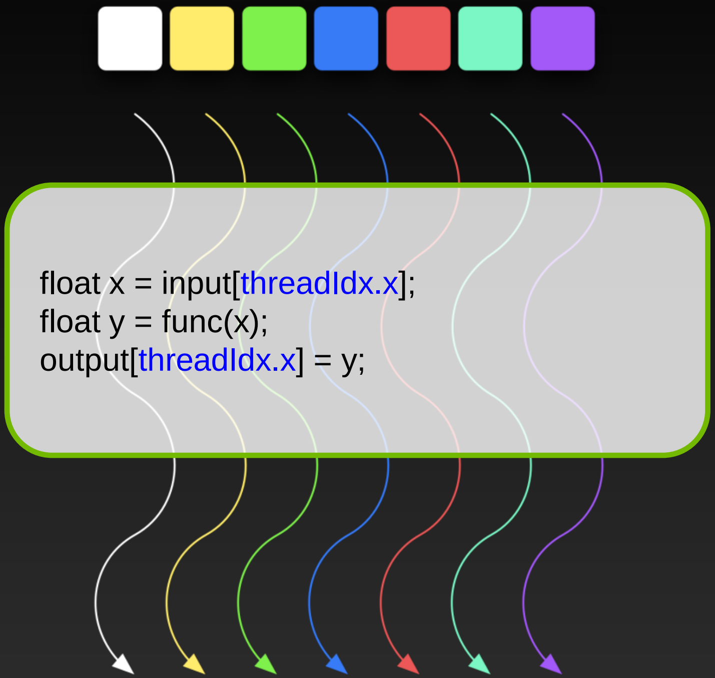
- Parallel portion of application: execute as a kernel
  - Entire GPU executes kernel, many threads

- CUDA threads:
  - Lightweight
  - Fast switching
  - 1000s execute simultaneously

CPU	Host	Executes functions
GPU	Device	Executes kernels

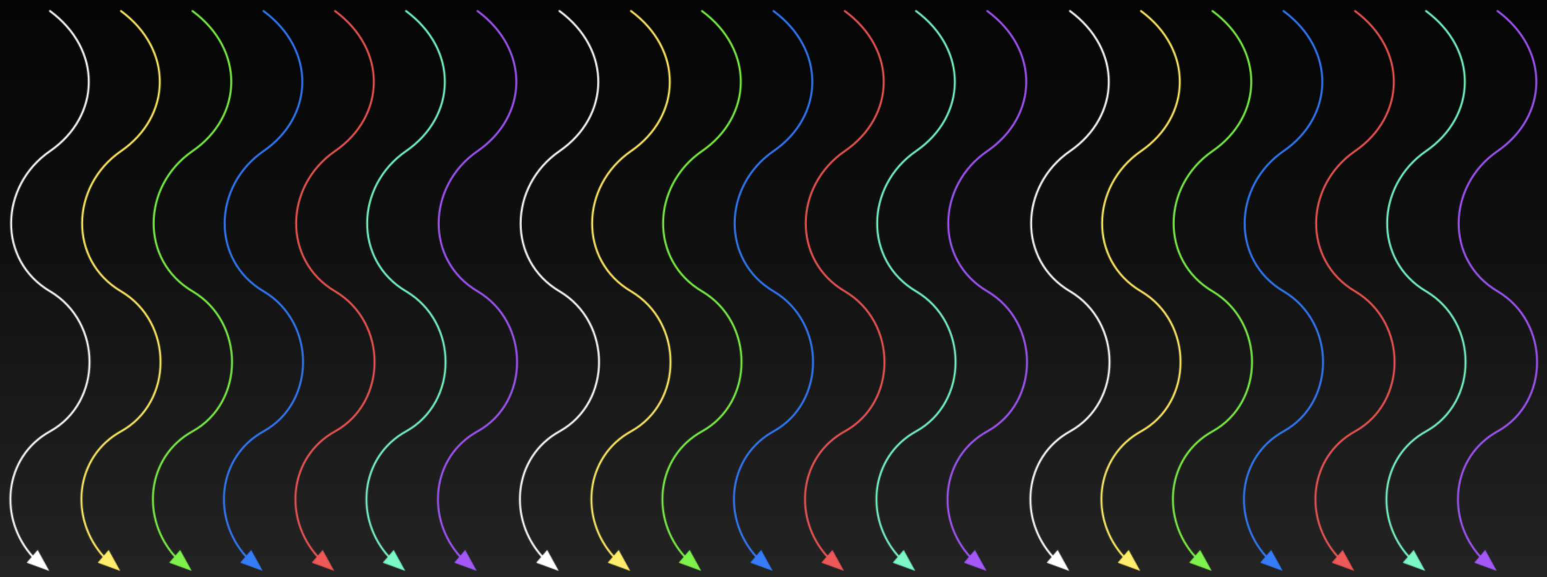
CUDA Kernels: Parallel Threads

- A kernel is a function executed on the GPU as an array of threads in parallel
- All threads execute the same code, can take different paths
- Each thread has an ID
  - Select input/output data
  - Control decisions

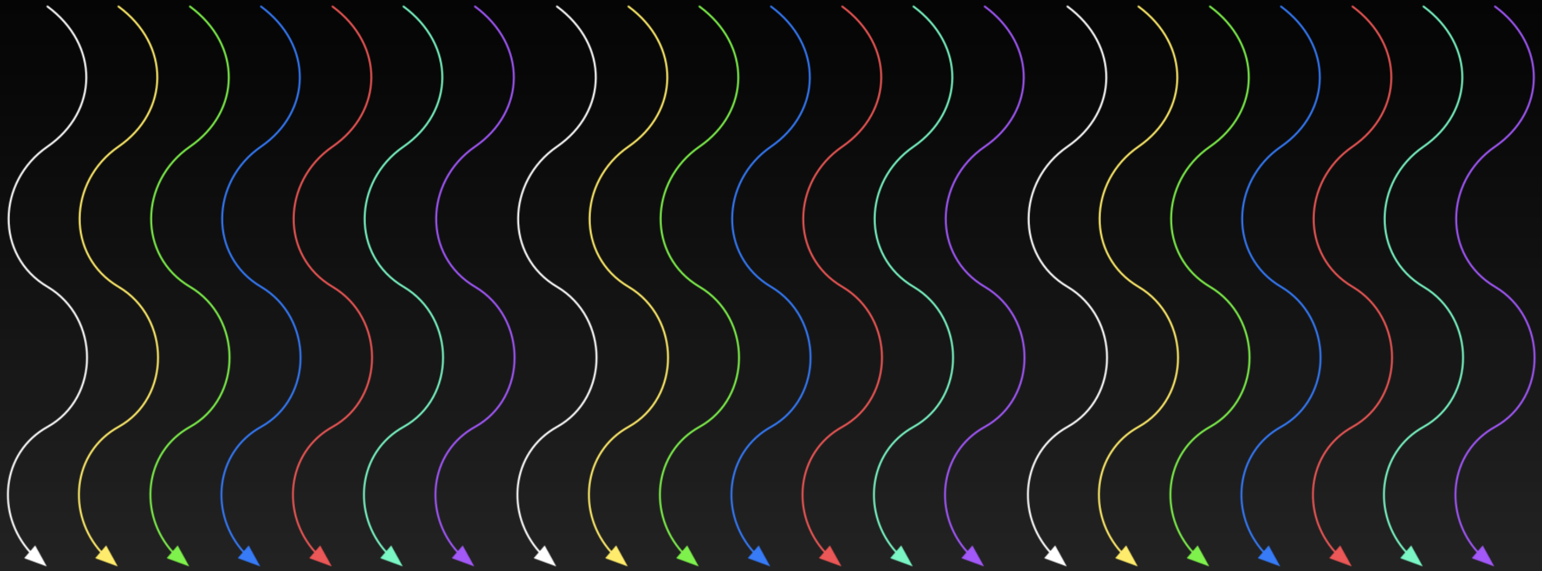


```
float x = input[threadIdx.x];
float y = func(x);
output[threadIdx.x] = y;
```

CUDA Kernels: Subdivide into Blocks

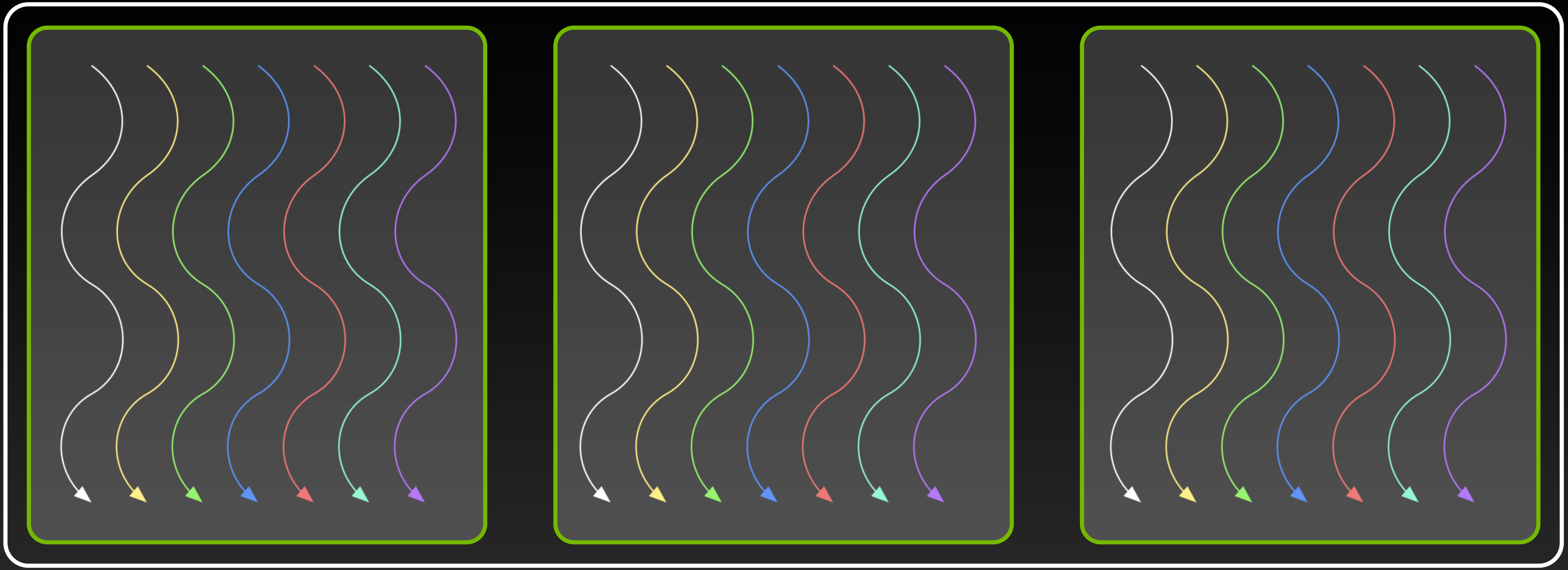


CUDA Kernels: Subdivide into Blocks



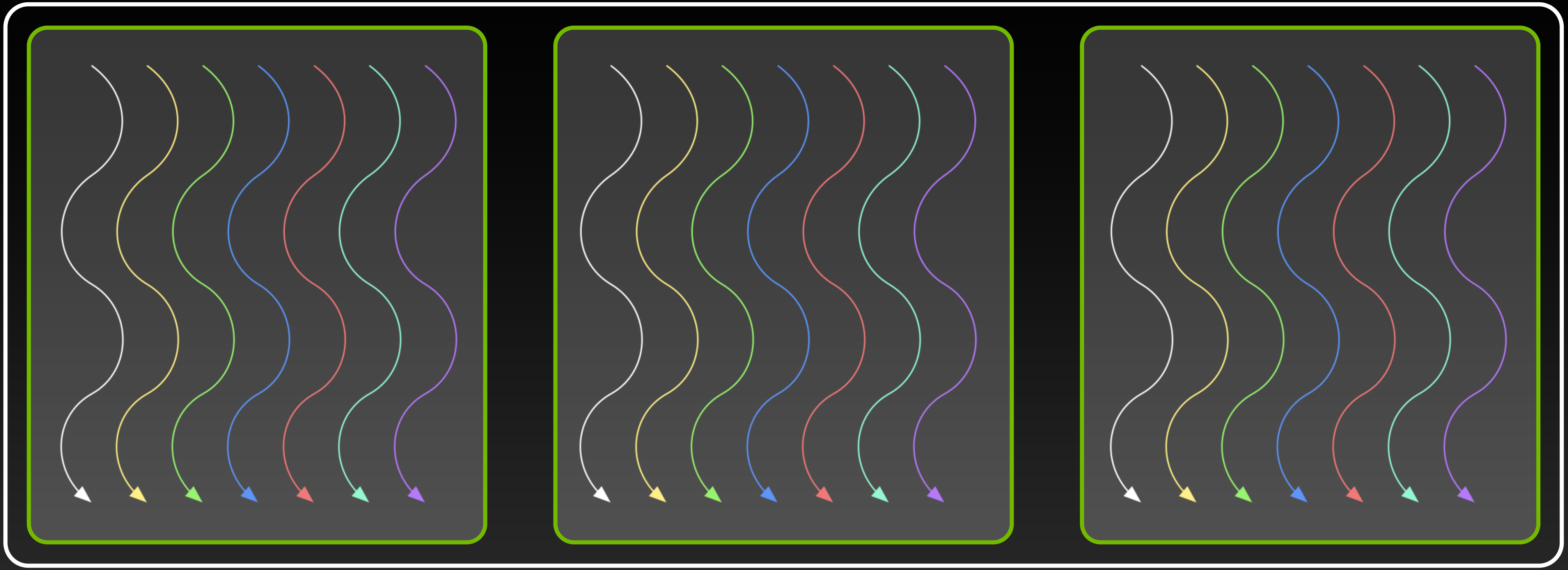
- Threads are grouped into blocks

CUDA Kernels: Subdivide into Blocks



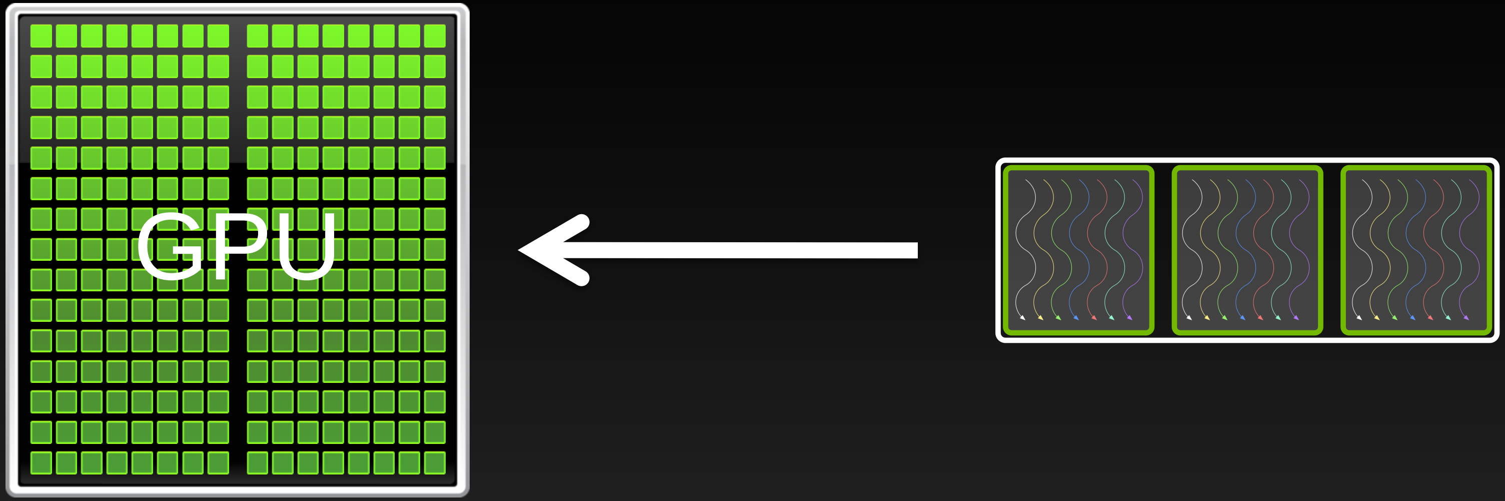
- Threads are grouped into **blocks**
- **Blocks** are grouped into **a grid**

CUDA Kernels: Subdivide into Blocks



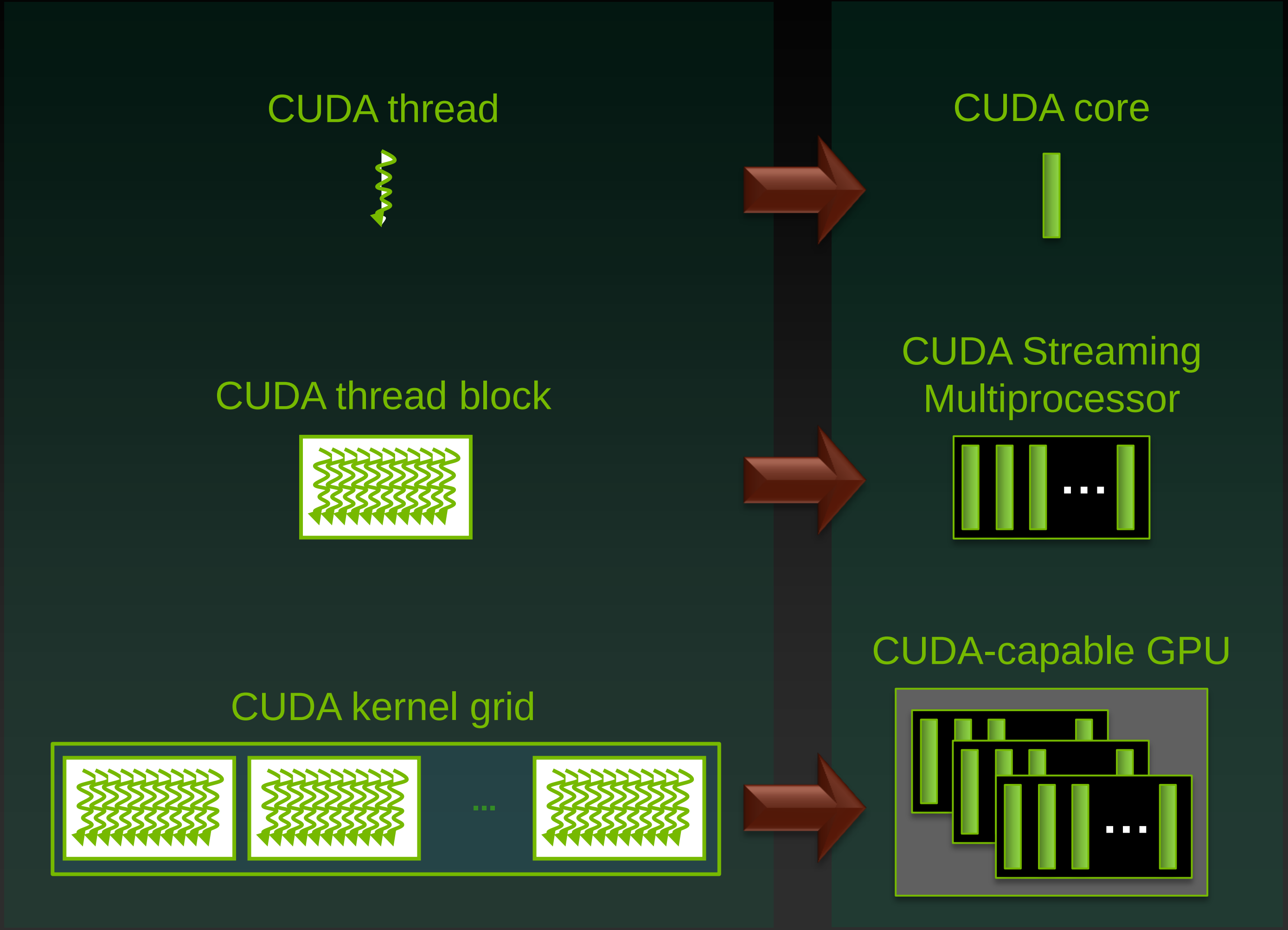
- Threads are grouped into blocks
- Blocks are grouped into a grid
- A kernel is executed as a grid of blocks of threads

CUDA Kernels: Subdivide into Blocks



- Threads are grouped into blocks
  - Note: Adjacent threads execute in lock-step scheduling groupings called warps; a block comprises one or more warps
- Blocks are grouped into a grid
- A kernel is executed as a grid of blocks of threads

Kernel Execution



- Each thread is executed by a core

- Each block is executed by one SM and does not migrate
- Several concurrent blocks can reside on one SM depending on the blocks' memory requirements and the SM's memory resources

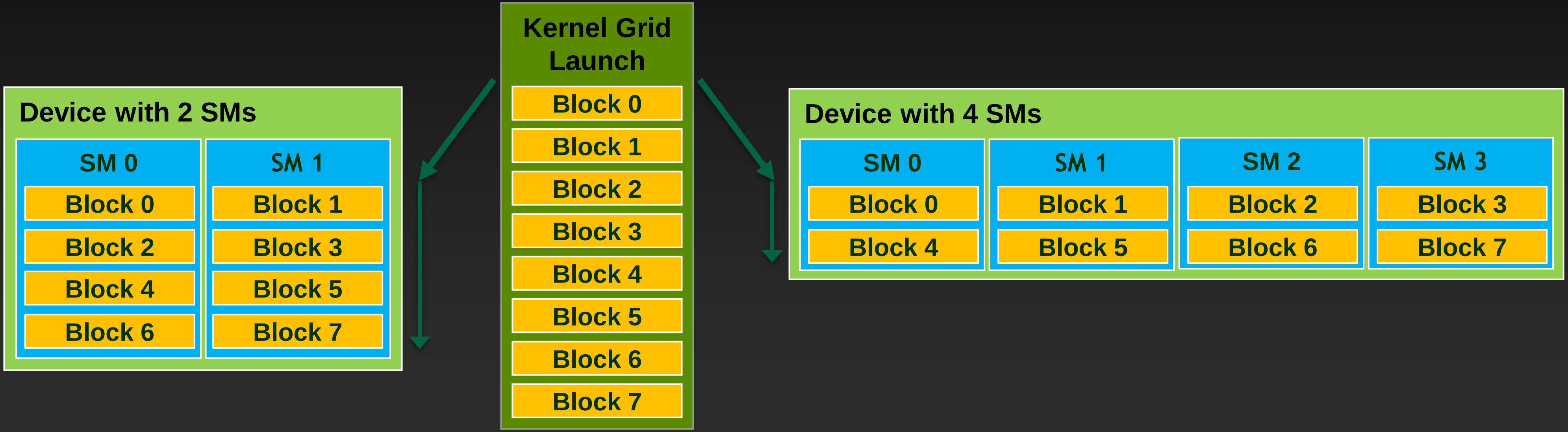
- Each kernel is executed on one device
- Multiple kernels can execute on a device at one time

Thread blocks allow cooperation

- Threads may need to cooperate:
  - Cooperatively load/store blocks of memory that they all use
  - Share results with each other or cooperate to produce a single result
  - Synchronize with each other

Thread blocks allow scalability

- Blocks can execute in any order, concurrently or sequentially
- This independence between blocks gives scalability:
  - A kernel scales across any number of SMs



MAPPING OPENACC TO CUDA

OpenACC Execution Model

- The OpenACC execution model has three levels: gang, worker, and vector

- Allows mapping to an architecture that is a collection of Processing Elements (PEs)
  - One or more PEs per node
  - Each PE is multi-threaded
  - Each thread can execute vector instructions

OpenACC Execution Model on CUDA

- The OpenACC execution model has three levels: gang, worker, and vector

- For GPUs, the mapping is implementation-dependent. Some possibilities:
  - gang==block, worker==warp, and vector==threads of a warp
  - omit “worker” and just have gang==block, vector==threads of a block

- Depends on what the compiler thinks is the best mapping for the problem

OpenACC Execution Model on CUDA

- The OpenACC execution model has three levels: gang, worker, and vector
- For GPUs, the mapping is implementation-dependent.
- ...But explicitly specifying that a given loop should map to gangs, workers, and/or vectors is optional anyway
  - Further specifying the *number* of gangs/workers/vectors is also optional
  - So why do it? To tune the code to fit a particular target architecture in a straightforward and easily re-tuned way.

PROFILING AND TUNING

Profiling Tools

- Use compiler output to determine how loops were mapped onto the accelerator
  - Not exactly “profiling”, but it’s helpful information that a GPU-aware profiler would also have given you
- PGI: Use PGI_ACC_TIME option to learn where time is being spent
- NVIDIA Visual Profiler
- 3rd-party profiling tools that are CUDA-aware
  - (But those are outside the scope of this talk)

PGI Accelerator compiler output

```
Accelerator kernel generated
57, #pragma acc loop gang, vector /* blockIdx.y threadIdx.y */
60, #pragma acc loop gang, vector /* blockIdx.x threadIdx.x */
CC 1.3 : 16 registers; 2112 shared, 40 constant, 0 local memory bytes; 100% occupancy
CC 2.0 : 19 registers; 2056 shared, 80 constant, 0 local memory bytes; 100% occupancy
```

CC stands for compute capability.
Fermi cards are 2.0, so we are looking at the second line

Number of registers used per thread. The max on Fermi is 63, so if you are close to that it might reduce occupancy

How many bytes of shared memory you are using per block. The compiler handles this, so you have no direct control over it.

If you need more than the maximum number of registers, then you will spill to local memory. Spilling is slow since it goes to (cached) global memory, so try to reduce register usage

Occupancy: how many threads will fit per SM vs. the max possible. If occupancy is low, try to see if you can reduce register or shared memory usage

PGI Accelerator profiling

- Compiler automatically instruments the code, outputs profile data
  - set PGI_ACC_TIME=1

```
Accelerator Kernel Timing data
./laplace2d.c
  main
    66: region entered 1000 times
        time(us): total=5515318 init=110 region=5515208
                  kernels=5320683 data=0
        w/o init: total=5515208 max=13486 min=5269 avg=5515
        70: kernel launched 1000 times
            grid: [16x512]  block: [32x8]
            time(us): total=5320683 max=5426 min=5200 avg=5320
./laplace2d.c
  main
    53: region entered 1000 times
        time(us): total=6493657 init=171 region=6493486
                  kernels=5108494 data=0
        ...
```

PGI Accelerator profiling

- Compiler automatically instruments the code, outputs profile data
- Provides insight into API-level efficiency
  - How many bytes of data were copied in and out?
  - How many times was each kernel launched, and how long did they take?
  - What kernel grid and block dimensions were used?

PGI Accelerator profiling

```
Total time: 13.874673 s

Accelerator Kernel Timing data
./laplace2d.c   main
    68: region entered 1000 times
        time(us): total=4903207 init=82 region=4903125
                  kernels=4852949 data=0
        w/o init: total=4903125 max=5109 min=4813 avg=4903
        71: kernel launched 1000 times
            grid: [256x256]  block: [16x16]
            time(us): total=4852949 max=5004 min=4769 avg=4852
./laplace2d.c  main
    56: region entered 1000 times
        time(us): total=8701161 init=57 region=8701104
                  kernels=8365523 data=0
        w/o init: total=8701104 max=8942 min=8638 avg=8701
        59: kernel launched 1000 times
            grid: [256x256]  block: [16x16]
            time(us): total=8222457 max=8310 min=8212 avg=8222
        63: kernel launched 1000 times
            grid: [1]  block: [256]
            time(us): total=143066 max=210 min=141 avg=143
./laplace2d.c  main
    50: region entered 1 time
        time(us): total=13874525 init=162566 region=13711959
                  data=64170
        w/o init: total=13711959 max=13711959 min=13711959 avg=13711959
```

Memcpy loop, taking 4.9s out of 13s

Main computation loop, taking 8.7s out of 13s

Enclosing while loop data region. Takes 13.7s, nearly the entire execution time

PGI Accelerator profiling

```
Total time: 13.874673 s

Accelerator Kernel Timing data
./laplace2d.c   main
    68: region entered 1000 times
        time(us): total=4903207 init=82 region=4903125
                  kernels=4852949 data=0
        w/o init: total=4903125 max=5109 min=4813 avg=4903
        71: kernel launched 1000 times
            grid: [256x256]  block: [16x16]
            time(us): total=4852949 max=5004 min=4769 avg=4852
./laplace2d.c  main
    56: region entered 1000 times
        time(us): total=8701161 init=57 region=8701104
                  kernels=8365523 data=0
        w/o init: total=8701104 max=8942 min=8638 avg=8701
        59: kernel launched 1000 times
            grid: [256x256]  block: [16x16]
            time(us): total=8222457 max=8310 min=8212 avg=8222
        63: kernel launched 1000 times
            grid: [1]  block: [256]
            time(us): total=143066 max=210 min=141 avg=143
./laplace2d.c  main
    50: region entered 1 time
        time(us): total=13874525 init=162566 region=13711959
                  data=64170
        w/o init: total=13711959 max=13711959 min=13711959 avg=13711959
```

Suboptimal grid and block dimensions

..how do we know this?

..how do we control it?

Mapping OpenACC to CUDA threads and blocks

```
#pragma acc kernels loop
   for( int i = 0; i < n; ++i ) y[i] += a*x[i];
```

Uses whatever mapping to threads and blocks the compiler chooses. Perhaps 16 blocks, 256 threads each

```
#pragma acc kernels loop gang(100), vector(128)
   for( int i = 0; i < n; ++i ) y[i] += a*x[i];
```

100 thread blocks, each with 128 threads, each thread executes one iteration of the loop, using kernels

```
#pragma acc parallel num_gangs(100), vector_length(128)
{
   #pragma acc loop gang, vector
   for( int i = 0; i < n; ++i ) y[i] += a*x[i];
}
```

100 thread blocks, each with 128 threads, each thread executes one iteration of the loop, using parallel

Mapping OpenACC to CUDA threads and blocks

```
#pragma acc parallel loop num_gangs(100)
{
   for( int i = 0; i < n; ++i ) y[i] += a*x[i];
}


#pragma acc parallel num_gangs(100)
{
   #pragma acc loop gang
   for( int i = 0; i < n; ++i ) y[i] += a*x[i];
}
```

100 thread blocks, each with apparently 1 thread, each thread redundantly executes the loop

Mapping OpenACC to CUDA threads and blocks

```
n = 12800;

#pragma acc kernels loop gang(100), vector(128)
   for( int i = 0; i < n; ++i ) y[i] += a*x[i];
```

100 thread blocks, each with 128 threads, each thread executes one iteration of the loop

```
#pragma acc kernels loop gang(50), vector(128)
   for( int i = 0; i < n; ++i ) y[i] += a*x[i];
```

50 thread blocks, each with 128 threads. Each thread does two elements worth of work

Doing multiple iterations per thread can improve performance by amortizing the cost of setup

Mapping OpenACC to CUDA threads and blocks

Nested loops generate multi-dimensional blocks and grids:

```
#pragma acc kernels loop gang(100), vector(16)
    for( … )

    #pragma acc loop gang(200), vector(32)
        for( … )
```

100 blocks tall (row/Y direction)

16 thread tall block

200 blocks wide (column/X direction)

32 thread wide block

Selecting block size (e.g., vectors per gang)

- Total number of threads in a block between 256 and 512 is usually a good number
  - Overly small blocks will limit the # of concurrent threads due to limitation on maximum # of concurrent blocks/SM
  - Overly large blocks can hinder performance, e.g., by increasing cost of any synchronizations/barrier among all the threads in a block
- All CUDA-capable GPUs to date prefer # threads per block to be a multiple of 32 if possible
  - …Since 32 threads is the *warp size* of current CUDA-capable GPUs
  - Non-multiples of 32 waste some resources and cycles
  - Furthermore, a multiple of 32 threads *wide* (x-dimension) is best (facilitates coalesced memory access to adjacent memory addresses)

Selecting block size (e.g., vectors per gang)

- Total number of threads in a block between 256 and 512 is usually a good number
- All CUDA-capable GPUs to date prefer # threads per block to be a multiple of 32 if possible
- So if we have 2D blocks, let's try a few combinations like 32x8, 64x4, 32x16, 64x8…

An aside on warps

- Blocks are divided into 32-thread-wide groups called warps
  - Size of warps is architecture-specific and can change in the future
- The SM creates, manages, schedules and executes threads at warp granularity
- All threads in a warp execute the same instruction at once
  - In case of divergence, the warp serially executes each branch path taken
- When accessing global memory, the accesses of the threads within a warp are coalesced into as few transactions as possible

Selecting grid size (e.g., number of gangs)

- Most obvious mapping is to have # of gangs times # of workers times # of vectors equal the total problem size
- We just saw that we can choose to manipulate this number so that each thread could do multiple pieces of work
  - Helps amortize the cost of setup for simple kernels
- What is the limit on how small we can/should go?
  - We at least want to have enough threads to fill the GPU several times over (perhaps 10 times or more), meaning we need 100,000+ threads.

PGI Accelerator compiler output

```
Accelerator kernel generated
57, #pragma acc loop gang, vector(8) /* blockIdx.y threadIdx.y */
60, #pragma acc loop gang(16), vector(32) /* blockIdx.x threadIdx.x */
CC 1.3 : 16 registers; 2112 shared, 40 constant, 0 local memory bytes; 100% occupancy
CC 2.0 : 19 registers; 2056 shared, 80 constant, 0 local memory bytes; 100% occupancy
```

Notice the compiler helpfully told us the mapping of gangs/vectors to blocks/threads that was used

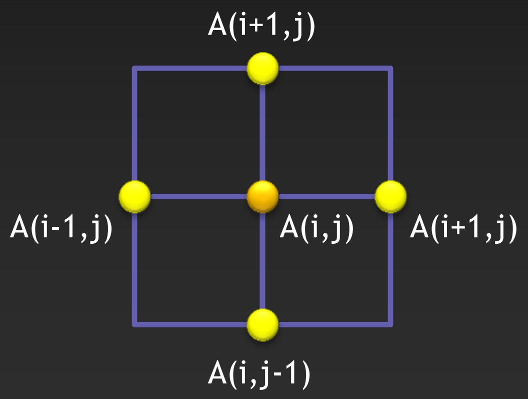
PGI Accelerator profiling: Recall earlier example…

```
Total time: 13.874673 s

Accelerator Kernel Timing data
./laplace2d.c   main
    68: region entered 1000 times
        time(us): total=4903207 init=82 region=4903125
                  kernels=4852949 data=0
        w/o init: total=4903125 max=5109 min=4813 avg=4903
        71: kernel launched 1000 times
            grid: [256x256]  block: [16x16]
            time(us): total=4852949 max=5004 min=4769 avg=4852
./laplace2d.c  main
    56: region entered 1000 times
        time(us): total=8701161 init=57 region=8701104
                  kernels=8365523 data=0
        w/o init: total=8701104 max=8942 min=8638 avg=8701
        59: kernel launched 1000 times
            grid: [256x256]  block: [16x16]
            time(us): total=8222457 max=8310 min=8212 avg=8222
        63: kernel launched 1000 times
            grid: [1]  block: [256]
            time(us): total=143066 max=210 min=141 avg=143
./laplace2d.c  main
    50: region entered 1 time
        time(us): total=13874525 init=162566 region=13711959
                  data=64170
        w/o init: total=13711959 max=13711959 min=13711959 avg=13711959
```

Example: Jacobi Iteration

- Iteratively converges to correct value (e.g. Temperature), by computing new values at each point from the average of neighboring points.
  - Common, useful algorithm
  - Example: Solve Laplace equation in 2D: $\nabla^2 f(x, y) = 0$



$$A_{k+1}(i,j) = \frac{A_k(i-1,j) + A_k(i+1,j) + A_k(i,j-1) + A_k(i,j+1)}{4}$$

Jacobi Iteration

- Task: use knowledge of GPU architecture to improve performance by specifying gang and vector clauses

Jacobi Iteration: OpenACC C v1

```
#pragma acc data copy(A), create(Anew)
while ( err > tol && iter < iter_max ) {
  err=0.0;

#pragma acc kernels loop reduction(max:err)
  for( int j = 1; j < n-1; j++) {
    for(int i = 1; i < m-1; i++) {

      Anew[j][i] = 0.25 * (A[j][i+1] + A[j][i-1] +
                           A[j-1][i] + A[j+1][i]);

      err = max(err, fabs(Anew[j][i] - A[j][i]);
    }
  }

#pragma acc kernels loop
  for( int j = 1; j < n-1; j++) {
    for( int i = 1; i < m-1; i++ ) {
      A[j][i] = Anew[j][i];
    }
  }

  iter++;
}
```

Jacobi Iteration: OpenACC C v2

```
#pragma acc data copy(A), create(Anew)
while ( err > tol && iter < iter_max ) {
  err=0.0;

#pragma acc kernels loop reduction(max:err)
  for( int j = 1; j < n-1; j++) {
#pragma acc loop gang(16) vector(32)
    for(int i = 1; i < m-1; i++) {

      Anew[j][i] = 0.25 * (A[j][i+1] + A[j][i-1] +
                           A[j-1][i] + A[j+1][i]);

      err = max(err, fabs(Anew[j][i] - A[j][i]);
    }
  }
#pragma acc kernels loop
  for( int j = 1; j < n-1; j++) {
#pragma acc loop gang(16) vector(32)
    for( int i = 1; i < m-1; i++ ) {
      A[j][i] = Anew[j][i];
    }
  }
  iter++;
}
```

Leave compiler to choose Y dimension for grids and blocks.

Grids are 16 blocks wide, blocks are 32 threads wide

Leave compiler to choose Y dimension for grids and blocks.

Grids are 16 blocks wide, blocks are 32 threads wide

Jacobi Iteration: OpenACC Fortran v1

```
!$acc data copy(A), create(Anew)
do while ( err > tol .and. iter < iter_max )
  err=0._fp_kind

!$acc kernels loop reduction(max:err)
  do j=1,m
    do i=1,n

      Anew(i,j) = .25_fp_kind * (A(i+1, j  ) + A(i-1, j  ) + &
                                 A(i  , j-1) + A(i  , j+1))

      err = max(err, Anew(i,j) - A(i,j))
    end do
  end do
!$acc end kernels

  ...

iter = iter +1
end do
!$acc end data
```

Jacobi Iteration: OpenACC Fortran v2

```
!$acc data copy(A), create(Anew)
do while ( err > tol .and. iter < iter_max )
  err=0._fp_kind

!$acc kernels loop reduction(max:err)
  do j=1,m
!$acc loop gang(16), vector(32)
    do i=1,n

      Anew(i,j) = .25_fp_kind * (A(i+1, j  ) + A(i-1, j  ) + &
                                 A(i  , j-1) + A(i  , j+1))

      err = max(err, Anew(i,j) - A(i,j))
    end do
  end do
!$acc end kernels

  ...

  iter = iter +1
end do
!$acc end data
```

Leave compiler to choose Y dimension for grids and blocks.

Grids are 16 blocks wide, blocks are 32 threads wide

Jacobi Iteration: Tune and Re-profile

- After modifying source code to specify number of gangs/vectors (i.e., grid/block dimensions), run again with PGI_ACC_TIME=1

```
total: 11.135176 s

./laplace2d.c
  main
    56: region entered 1000 times
        time(us): total=5568043 init=68 region=5567975
                  kernels=5223007 data=0
        w/o init: total=5567975 max=6040 min=5464 avg=5567
        60: kernel launched 1000 times
            grid: [16x512]  block: [32x8]
            time(us): total=5197462 max=5275 min=5131 avg=5197
        64: kernel launched 1000 times
            grid: [1]  block: [256]
            time(us): total=25545 max=119 min=24 avg=25
```

Main computation loop performance improved from 8.7s to 5.5s

Grid size changed from [256x256] to [16x512]

Block size changed from [16x16] to [32x8]

Performance: v1

CPU: Intel Xeon X5680
6 Cores @ 3.33GHz

GPU: NVIDIA Tesla M2070

Execution	Time (s)	Speedup
CPU 1 OpenMP thread	69.80	--
CPU 2 OpenMP threads	44.76	1.56x
CPU 4 OpenMP threads	39.59	1.76x
CPU 6 OpenMP threads	39.71	1.76x
OpenACC GPU	13.65	2.9x

Speedup vs. 1 CPU core

Speedup vs. 6 CPU cores

Note: same code runs in 9.78s on NVIDIA Tesla M2090 GPU

Performance: v2

CPU: Intel Xeon X5680
6 Cores @ 3.33GHz

GPU: NVIDIA Tesla M2070

Execution	Time (s)	Speedup
CPU 1 OpenMP thread	69.80	--
CPU 2 OpenMP threads	44.76	1.56x
CPU 4 OpenMP threads	39.59	1.76x
CPU 6 OpenMP threads	39.71	1.76x
OpenACC GPU	10.98	3.62x

Speedup vs. 1 CPU core

Speedup vs. 6 CPU cores

Note: same code runs in 7.58s on NVIDIA Tesla M2090 GPU

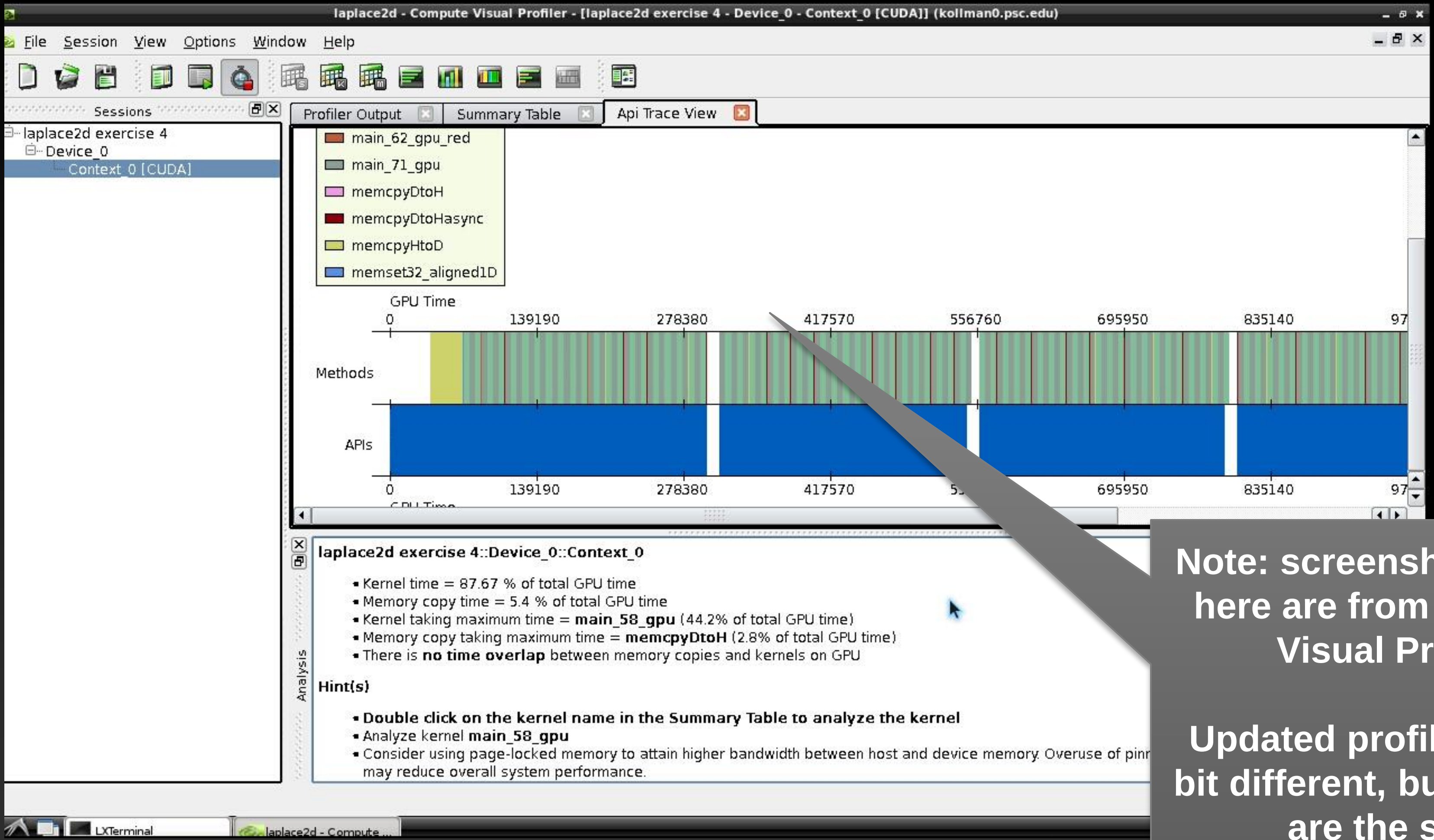
PGI Accelerator profiling

- Compiler automatically instruments the code, outputs profile data
- Provides insight into API-level efficiency
  - How many bytes of data were copied in and out?
  - How many times was each kernel launched, and how long did they take?
  - What kernel grid and block dimensions were used?
- Clearly this can help us to tune to fit the architecture better
- What if we want to go even deeper?
  - Compiler instrumentation provides relatively little insight (at present) into how efficient the kernels themselves were

Profiling Tools

- Need a profiling tool that is more aware of the inner workings of the GPU to provide deeper insights
- E.g.: NVIDIA Visual Profiler

NVIDIA Visual Profiler



laplace2d exercise 4::Device_0::Context_0

- Kernel time = 87.67 % of total GPU time
- Memory copy time = 5.4 % of total GPU time
- Kernel taking maximum time = **main_58_gpu** (44.2% of total GPU time)
- Memory copy taking maximum time = **memcpyDtoH** (2.8% of total GPU time)
- There is **no time overlap** between memory copies and kernels on GPU

Hint(s)

- **Double click on the kernel name in the Summary Table to analyze the kernel**
- Analyze kernel **main_58_gpu**
- Consider using page-locked memory to attain higher bandwidth between host and device memory. Overuse of pinn
  may reduce overall system performance.

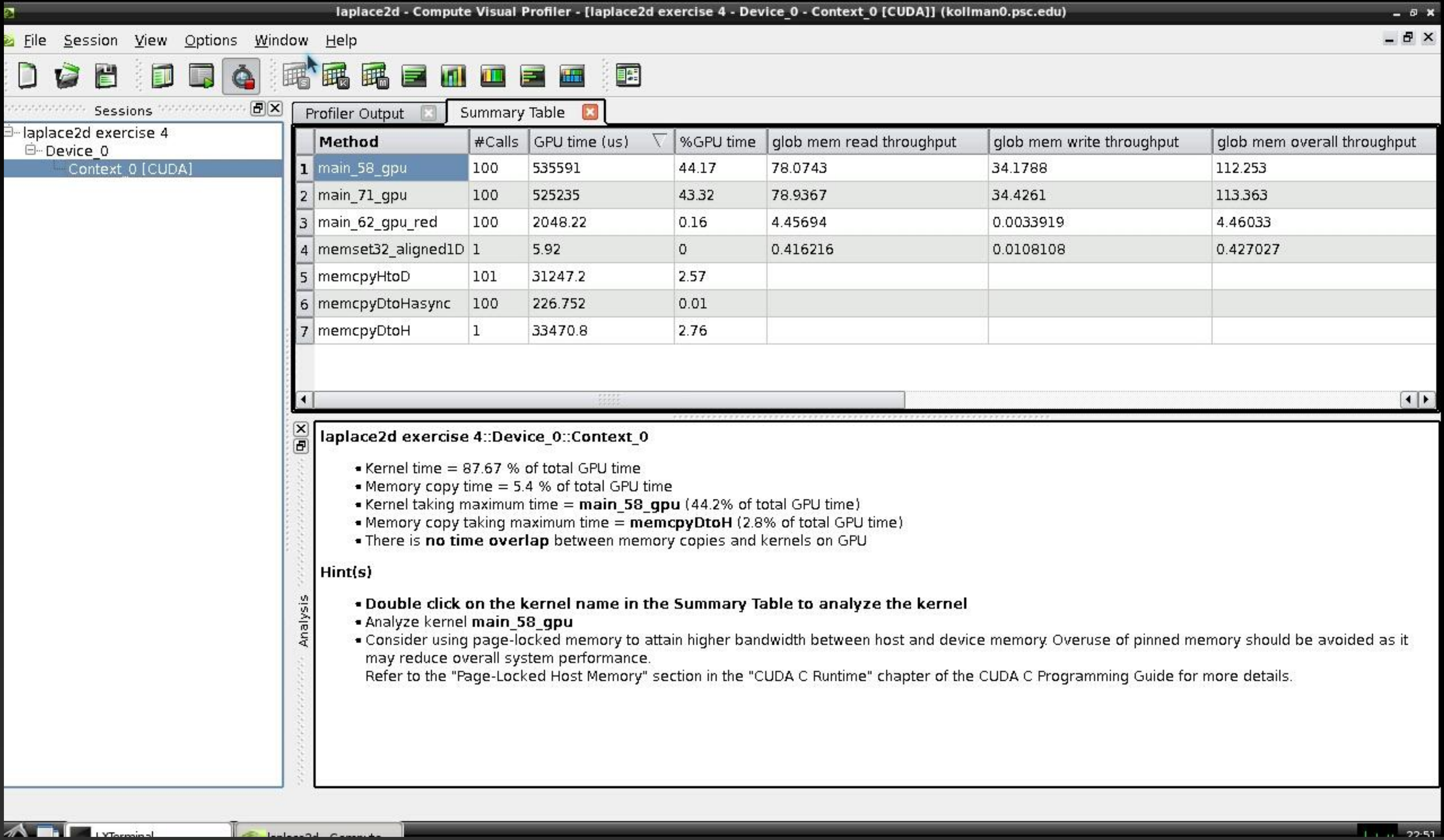
Note: screenshots shown here are from CUDA 4.0 Visual Profiler

Updated profiler looks a bit different, but concepts are the same

Jacobi Iteration: Kernel Profiling

- Task: use NVIDIA Visual Profiler data to identify additional optimization opportunities in Jacobi example

Jacobi Iteration: Kernel Profiling


laplace2d - Compute Visual Profiler - [laplace2d exercise 4 - Device_0 - Context_0 [CUDA]] (kollman0.psc.edu)
File Session View Options Window Help
Sessions
laplace2d exercise 4
Device_0
Context_0 [CUDA]
Profiler Output
Summary Table
Analysis

	Method	#Calls	GPU time (us)	%GPU time	glob mem read throughput	glob mem write throughput	glob mem overall throughput
1	main_58_gpu	100	535591	44.17	78.0743	34.1788	112.253
2	main_71_gpu	100	525235	43.32	78.9367	34.4261	113.363
3	main_62_gpu_red	100	2048.22	0.16	4.45694	0.0033919	4.46033
4	memset32_aligned1D	1	5.92	0	0.416216	0.0108108	0.427027
5	memcpyHtoD	101	31247.2	2.57			
6	memcpyDtoHasync	100	226.752	0.01			
7	memcpyDtoH	1	33470.8	2.76			

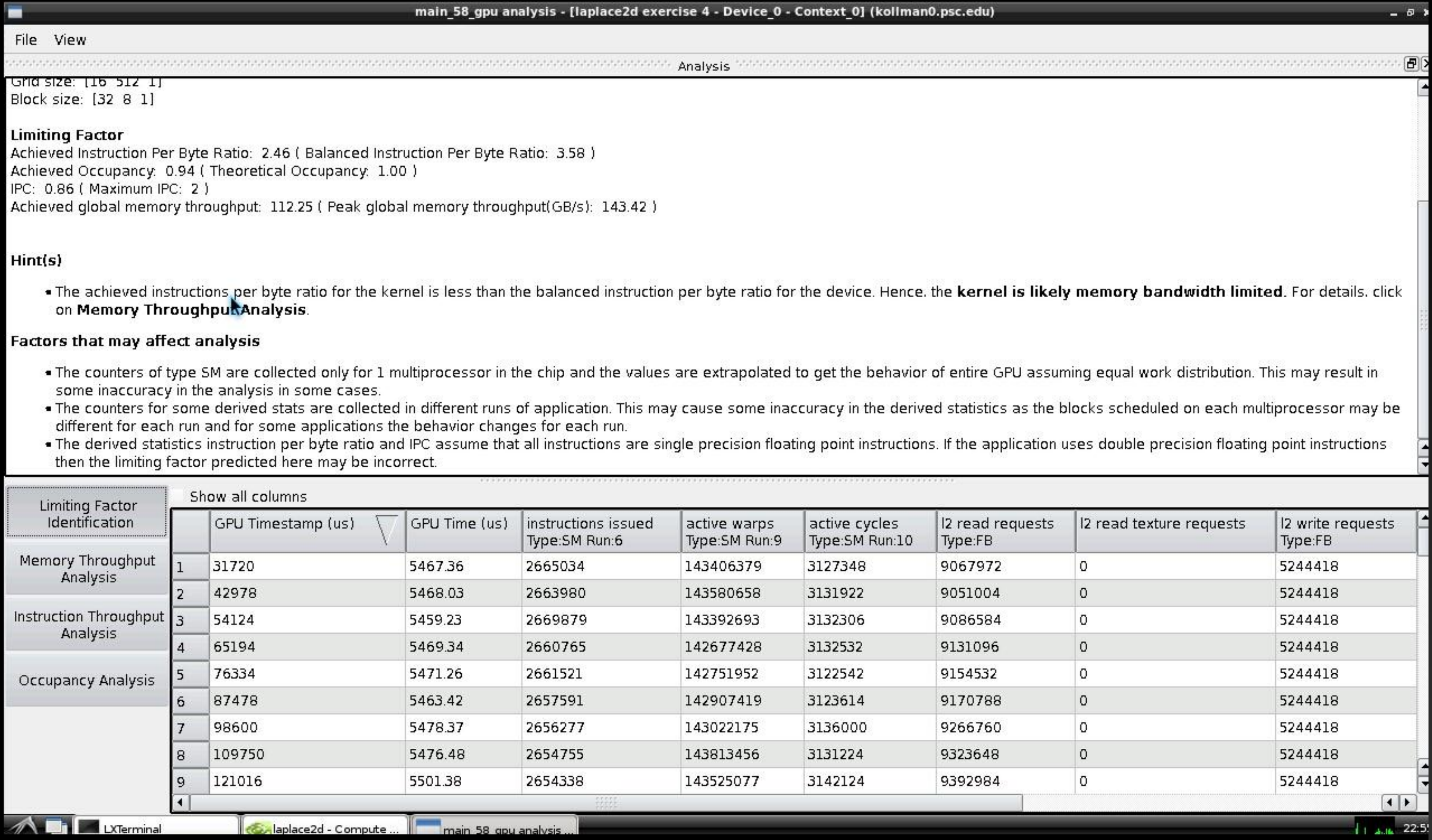
laplace2d exercise 4::Device_0::Context_0

- Kernel time = 87.67 % of total GPU time
- Memory copy time = 5.4 % of total GPU time
- Kernel taking maximum time = **main_58_gpu** (44.2% of total GPU time)
- Memory copy taking maximum time = **memcpyDtoH** (2.8% of total GPU time)
- There is **no time overlap** between memory copies and kernels on GPU

Hint(s)

- **Double click on the kernel name in the Summary Table to analyze the kernel**
- Analyze kernel **main_58_gpu**
- Consider using page-locked memory to attain higher bandwidth between host and device memory. Overuse of pinned memory should be avoided as it may reduce overall system performance.
  Refer to the "Page-Locked Host Memory" section in the "CUDA C Runtime" chapter of the CUDA C Programming Guide for more details.

Jacobi Iteration: Kernel Profiling



main_58_gpu analysis - [laplace2d exercise 4 - Device_0 - Context_0] (kollman0.psc.edu)

Grid size: [16 512 1]
Block size: [32 8 1]

Limiting Factor
Achieved Instruction Per Byte Ratio: 2.46 (Balanced Instruction Per Byte Ratio: 3.58)
Achieved Occupancy: 0.94 (Theoretical Occupancy: 1.00)
IPC: 0.86 (Maximum IPC: 2)
Achieved global memory throughput: 112.25 (Peak global memory throughput(GB/s): 143.42)

Hint(s)

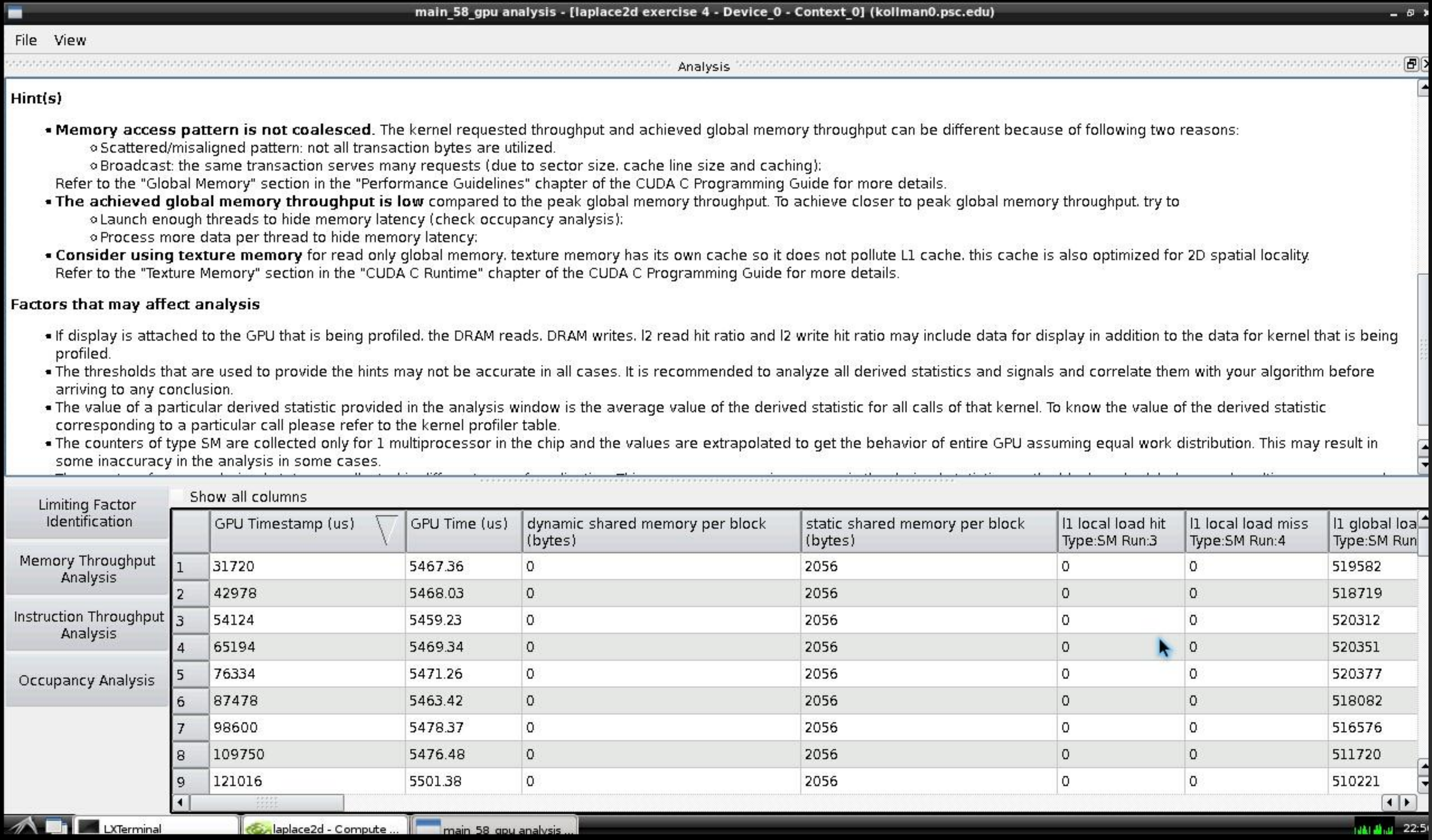
- The achieved instructions per byte ratio for the kernel is less than the balanced instruction per byte ratio for the device. Hence, the **kernel is likely memory bandwidth limited.** For details, click on **Memory Throughput Analysis**.

Factors that may affect analysis

- The counters of type SM are collected only for 1 multiprocessor in the chip and the values are extrapolated to get the behavior of entire GPU assuming equal work distribution. This may result in some inaccuracy in the analysis in some cases.
- The counters for some derived stats are collected in different runs of application. This may cause some inaccuracy in the derived statistics as the blocks scheduled on each multiprocessor may be different for each run and for some applications the behavior changes for each run.
- The derived statistics instruction per byte ratio and IPC assume that all instructions are single precision floating point instructions. If the application uses double precision floating point instructions then the limiting factor predicted here may be incorrect.

	GPU Timestamp (us)	GPU Time (us)	instructions issued Type:SM Run:6	active warps Type:SM Run:9	active cycles Type:SM Run:10	l2 read requests Type:FB	l2 read texture requests	l2 write requests Type:FB
1	31720	5467.36	2665034	143406379	3127348	9067972	0	5244418
2	42978	5468.03	2663980	143580658	3131922	9051004	0	5244418
3	54124	5459.23	2669879	143392693	3132306	9086584	0	5244418
4	65194	5469.34	2660765	142677428	3132532	9131096	0	5244418
5	76334	5471.26	2661521	142751952	3122542	9154532	0	5244418
6	87478	5463.42	2657591	142907419	3123614	9170788	0	5244418
7	98600	5478.37	2656277	143022175	3136000	9266760	0	5244418
8	109750	5476.48	2654755	143813456	3131224	9323648	0	5244418
9	121016	5501.38	2654338	143525077	3142124	9392984	0	5244418

Jacobi Iteration: Kernel Profiling


main_58_gpu analysis - [laplace2d exercise 4 - Device_0 - Context_0] (kollman0.psc.edu)
File
View
Analysis

Hint(s)

- **Memory access pattern is not coalesced.** The kernel requested throughput and achieved global memory throughput can be different because of following two reasons:
  - Scattered/misaligned pattern: not all transaction bytes are utilized.
  - Broadcast: the same transaction serves many requests (due to sector size, cache line size and caching);

  Refer to the "Global Memory" section in the "Performance Guidelines" chapter of the CUDA C Programming Guide for more details.
- **The achieved global memory throughput is low** compared to the peak global memory throughput. To achieve closer to peak global memory throughput, try to
  - Launch enough threads to hide memory latency (check occupancy analysis);
  - Process more data per thread to hide memory latency;
- **Consider using texture memory** for read only global memory, texture memory has its own cache so it does not pollute L1 cache, this cache is also optimized for 2D spatial locality.
  Refer to the "Texture Memory" section in the "CUDA C Runtime" chapter of the CUDA C Programming Guide for more details.

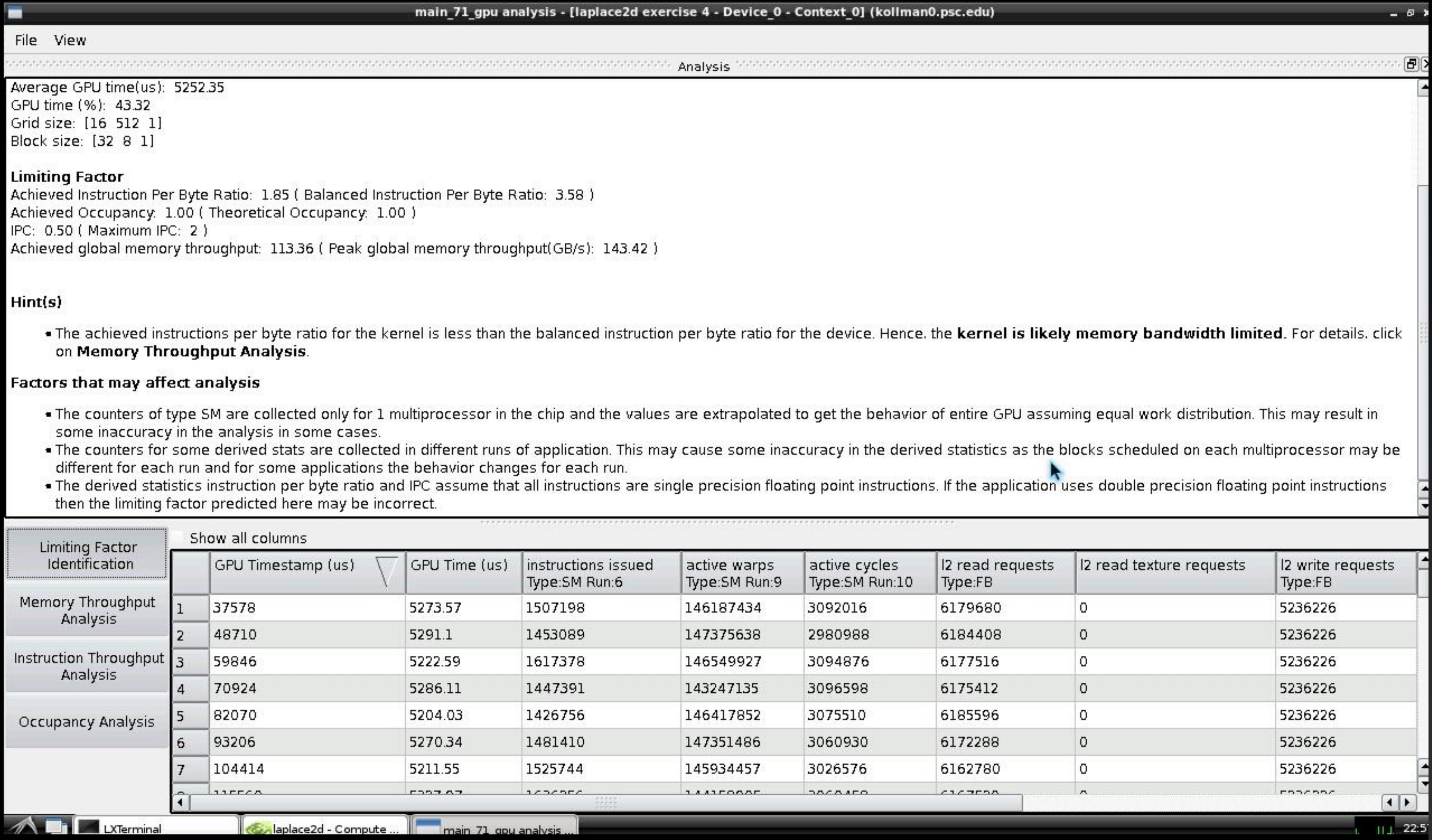
Factors that may affect analysis

- If display is attached to the GPU that is being profiled, the DRAM reads, DRAM writes, l2 read hit ratio and l2 write hit ratio may include data for display in addition to the data for kernel that is being profiled.
- The thresholds that are used to provide the hints may not be accurate in all cases. It is recommended to analyze all derived statistics and signals and correlate them with your algorithm before arriving to any conclusion.
- The value of a particular derived statistic provided in the analysis window is the average value of the derived statistic for all calls of that kernel. To know the value of the derived statistic corresponding to a particular call please refer to the kernel profiler table.
- The counters of type SM are collected only for 1 multiprocessor in the chip and the values are extrapolated to get the behavior of entire GPU assuming equal work distribution. This may result in some inaccuracy in the analysis in some cases.

Limiting Factor Identification
Memory Throughput Analysis
Instruction Throughput Analysis
Occupancy Analysis
Show all columns

	GPU Timestamp (us)	GPU Time (us)	dynamic shared memory per block (bytes)	static shared memory per block (bytes)	l1 local load hit Type:SM Run:3	l1 local load miss Type:SM Run:4	l1 global loa Type:SM Run
1	31720	5467.36	0	2056	0	0	519582
2	42978	5468.03	0	2056	0	0	518719
3	54124	5459.23	0	2056	0	0	520312
4	65194	5469.34	0	2056	0	0	520351
5	76334	5471.26	0	2056	0	0	520377
6	87478	5463.42	0	2056	0	0	518082
7	98600	5478.37	0	2056	0	0	516576
8	109750	5476.48	0	2056	0	0	511720
9	121016	5501.38	0	2056	0	0	510221

Jacobi Iteration: Kernel Profiling



main_71_gpu analysis - [laplace2d exercise 4 - Device_0 - Context_0] (kollman0.psc.edu)

File View

Analysis

Average GPU time(us): 5252.35
GPU time (%): 43.32
Grid size: [16 512 1]
Block size: [32 8 1]

Limiting Factor
Achieved Instruction Per Byte Ratio: 1.85 (Balanced Instruction Per Byte Ratio: 3.58)
Achieved Occupancy: 1.00 (Theoretical Occupancy: 1.00)
IPC: 0.50 (Maximum IPC: 2)
Achieved global memory throughput: 113.36 (Peak global memory throughput(GB/s): 143.42)

Hint(s)

- The achieved instructions per byte ratio for the kernel is less than the balanced instruction per byte ratio for the device. Hence, the **kernel is likely memory bandwidth limited**. For details, click on **Memory Throughput Analysis**.

Factors that may affect analysis

- The counters of type SM are collected only for 1 multiprocessor in the chip and the values are extrapolated to get the behavior of entire GPU assuming equal work distribution. This may result in some inaccuracy in the analysis in some cases.
- The counters for some derived stats are collected in different runs of application. This may cause some inaccuracy in the derived statistics as the blocks scheduled on each multiprocessor may be different for each run and for some applications the behavior changes for each run.
- The derived statistics instruction per byte ratio and IPC assume that all instructions are single precision floating point instructions. If the application uses double precision floating point instructions then the limiting factor predicted here may be incorrect.

	GPU Timestamp (us)	GPU Time (us)	instructions issued Type:SM Run:6	active warps Type:SM Run:9	active cycles Type:SM Run:10	l2 read requests Type:FB	l2 read texture requests	l2 write requests Type:FB
1	37578	5273.57	1507198	146187434	3092016	6179680	0	5236226
2	48710	5291.1	1453089	147375638	2980988	6184408	0	5236226
3	59846	5222.59	1617378	146549927	3094876	6177516	0	5236226
4	70924	5286.11	1447391	143247135	3096598	6175412	0	5236226
5	82070	5204.03	1426756	146417852	3075510	6185596	0	5236226
6	93206	5270.34	1481410	147351486	3060930	6172288	0	5236226
7	104414	5211.55	1525744	145934457	3026576	6162780	0	5236226
8	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]

Jacobi Iteration: OpenACC C v2

```
#pragma acc data copy(A), create(Anew)
while ( err > tol && iter < iter_max ) {
  err=0.0;

#pragma acc kernels loop reduction(max:err)
  for( int j = 1; j < n-1; j++) {
#pragma acc loop gang(16) vector(32)
    for(int i = 1; i < m-1; i++) {

      Anew[j][i] = 0.25 * (A[j][i+1] + A[j][i-1] +
                           A[j-1][i] + A[j+1][i]);

      err = max(err, fabs(Anew[j][i] - A[j][i]);
    }
  }
#pragma acc kernels loop
  for( int j = 1; j < n-1; j++) {
#pragma acc loop gang(16) vector(32)
    for( int i = 1; i < m-1; i++ ) {
      A[j][i] = Anew[j][i];
    }
  }
  iter++;
}
```

Jacobi Iteration: OpenACC C v3

```
#pragma acc data copy(A), copyin(Anew)
while ( err > tol && iter < iter_max ) {
  err=0.0;

#pragma acc kernels loop
  for( int j = 1; j < n-1; j++) {
#pragma acc loop gang(16) vector(32)
    for(int i = 1; i < m-1; i++) {

      Anew[j][i] = 0.25 * (A[j][i+1] + A[j][i-1] +
                           A[j-1][i] + A[j+1][i]);
    }
  }
#pragma acc kernels loop reduction(max:err)
  for( int j = 1; j < n-1; j++) {
#pragma acc loop gang(16) vector(32)
    for( int i = 1; i < m-1; i++ ) {
      A[j][i] = 0.25 * (Anew[j][i+1] + Anew[j][i-1] +
                        Anew[j-1][i] + Anew[j+1][i]);
      err = max(err, fabs(A[j][i] - Anew[j][i]);
    }
  }
  iter+=2;
}
```

Need to switch back to copying Anew in to accelerator so that halo cells will be correct

Can calculate the max reduction on 'error' once per pair, so removed it from this loop

Replace memcpy kernel with a second instance of the stencil kernel

Only need half as many times through the loop now

Performance

CPU: Intel Xeon X5680
6 Cores @ 3.33GHz

GPU: NVIDIA Tesla M2070

Execution	Time (s)	Speedup
CPU 6 OpenMP threads (v2)	39.7	
CPU 6 OpenMP threads (v3)	20.4	1.95x
OpenACC GPU (v2)	11.0	
OpenACC GPU (v3)	5.7	1.93x

Next Steps and Further Information

- Stay tuned for Part 3: Advanced OpenACC
- Later this week:
  - Sessions on profiling tools
  - Sessions on CUDA performance tuning and analysis
- Reading material:
  - CUDA C Programming Guide
  - CUDA C Best Practices Guide

GPU TECHNOLOGY CONFERENCE

Questions?