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
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
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

<table>
<thead>
<tr>
<th></th>
<th>Host</th>
<th>Device</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>GPU</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Executes functions</td>
<td>Executes kernels</td>
<td></td>
</tr>
</tbody>
</table>
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

```plaintext
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
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
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

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=5104 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
.

Suboptimal grid and block dimensions

..how do we know this?

..how do we control it?
Mapping OpenACC to CUDA threads and blocks

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

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

#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];
}
```

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

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

100 thread blocks, each with 128 threads, each thread executes one iteration of the loop, using parallel.
Mapping OpenACC to CUDA threads and blocks

```c
#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.
n = 12800;

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

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

Doing multiple iterations per thread can improve performance by amortizing the cost of setup.
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( ... )
```

Mapping OpenACC to CUDA threads and blocks:
- 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.
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=5109 min=4813 avg=4903

./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=8942 min=8638 avg=8701
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: \( \nabla^2 f(x, y) = 0 \)

\[
A_{k+1}(i, j) = \frac{A_k(i - 1, j) + A_k(i + 1, j) + A_k(i, j - 1) + A_k(i, j + 1)}{4}
\]
Task: use knowledge of GPU architecture to improve performance by specifying gang and vector clauses
Jacobi Iteration: OpenACC C v1

```c
#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++ ) {
            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

```c
#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++ ) {
            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 Fortran v1

```fortran
!$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

...
Jacobi Iteration: OpenACC Fortran v2

```fortran
!$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/elements (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  

<table>
<thead>
<tr>
<th>Execution</th>
<th>Time (s)</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU 1 OpenMP thread</td>
<td>69.80</td>
<td>--</td>
</tr>
<tr>
<td>CPU 2 OpenMP threads</td>
<td>44.76</td>
<td>1.56x</td>
</tr>
<tr>
<td>CPU 4 OpenMP threads</td>
<td>39.59</td>
<td>1.76x</td>
</tr>
<tr>
<td>CPU 6 OpenMP threads</td>
<td>39.71</td>
<td>1.76x</td>
</tr>
<tr>
<td>OpenACC GPU</td>
<td>13.65</td>
<td>2.9x</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Execution</th>
<th>Time (s)</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU 1 OpenMP thread</td>
<td>69.80</td>
<td>--</td>
</tr>
<tr>
<td>CPU 2 OpenMP threads</td>
<td>44.76</td>
<td>1.56x</td>
</tr>
<tr>
<td>CPU 4 OpenMP threads</td>
<td>39.59</td>
<td>1.76x</td>
</tr>
<tr>
<td>CPU 6 OpenMP threads</td>
<td>39.71</td>
<td>1.76x</td>
</tr>
<tr>
<td>OpenACC GPU</td>
<td>10.98</td>
<td>3.62x</td>
</tr>
</tbody>
</table>

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

Updated profiler looks a bit different, but concepts are the same.
Task: use NVIDIA Visual Profiler data to identify additional optimization opportunities in Jacobi example
Jacobi Iteration: Kernel Profiling

[Table with columns and rows showing various methods, their calls, GPU time, and various throughput metrics.]

**Notes:**
- Kernel time = 87.67% of total GPU time
- Memory copy time = 3.4% of total GPU time
- Kernel taking maximum time = main_58_gpu (44.2% of total GPU time)
- Memory copy taking maximum time = memcpyToD (2.8% of total GPU time)
- There is no time overlap between memory copies and kernels on GPU

**Hints:**
- 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

**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: 0.66 | Maximum IPC: 2
- Achieved global memory throughput: 112.25 | Peak global memory throughput (GB/s): 143.41

**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.

<table>
<thead>
<tr>
<th>Limiting Factor</th>
<th>Show all columns</th>
<th>GPU Time (us)</th>
<th>Instructions issued</th>
<th>active warps</th>
<th>active cycles</th>
<th>l2 read requests</th>
<th>l2 read texture requests</th>
<th>l2 write requests</th>
</tr>
</thead>
<tbody>
<tr>
<td>Identification</td>
<td></td>
<td></td>
<td>Type SM Run 0</td>
<td>Type SM Run 0</td>
<td>Type SM Run 0</td>
<td>Type SM Run 0</td>
<td>Type SM Run 0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>5467.36</td>
<td>2565024</td>
<td>143406379</td>
<td>3127348</td>
<td>9067972</td>
<td>0</td>
<td>5244118</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>5460.03</td>
<td>2663890</td>
<td>14358650</td>
<td>3131922</td>
<td>9051004</td>
<td>0</td>
<td>5244118</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>5459.23</td>
<td>2665878</td>
<td>14332629</td>
<td>3131296</td>
<td>9068564</td>
<td>0</td>
<td>5244118</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>5460.14</td>
<td>2665765</td>
<td>142677420</td>
<td>3132532</td>
<td>9131096</td>
<td>0</td>
<td>5244118</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td>5471.26</td>
<td>2661521</td>
<td>14275192</td>
<td>3122542</td>
<td>9154532</td>
<td>0</td>
<td>5244118</td>
</tr>
<tr>
<td>6</td>
<td></td>
<td>5462.42</td>
<td>2657391</td>
<td>142967419</td>
<td>3126124</td>
<td>9170798</td>
<td>0</td>
<td>5244118</td>
</tr>
<tr>
<td>7</td>
<td></td>
<td>5470.17</td>
<td>2656277</td>
<td>143022175</td>
<td>3136000</td>
<td>9266780</td>
<td>0</td>
<td>5244118</td>
</tr>
<tr>
<td>8</td>
<td></td>
<td>5476.48</td>
<td>2654753</td>
<td>143813456</td>
<td>3131224</td>
<td>9323648</td>
<td>0</td>
<td>5244118</td>
</tr>
<tr>
<td>9</td>
<td></td>
<td>5501.36</td>
<td>2654336</td>
<td>143525077</td>
<td>3142124</td>
<td>9392634</td>
<td>0</td>
<td>5244118</td>
</tr>
</tbody>
</table>
Jacobi Iteration: Kernel Profiling

### Memory Access Pattern
- **Memory access pattern is not coalesced.** The kernel requested throughput and achieved global memory throughput can be different because of following two reasons:
  - Scattered/aligned pattern not all transaction bytes are utilized.
  - Broadcast/store transaction many requests (due to thread block size, cache line size and caching).
- **Achieved global memory throughput is low** compared to the peak global memory throughput. To achieve closer to peak global memory throughput try to:
  - Launch more warps to hide memory latency.
  - Process more data per thread to hide memory latency.
- **Consider using texture memory for read only memory.** Texture memory has its own cache so it does not pollute L1 cache. This cache is also optimized for 2D spatial locality.

### Factors that may affect analysis
- If display is attached to the GPU that is being profiled, the DRAM reads/DRAM writes. IZ read hit ratio and IZ write hit ratio may include data for display in addition to the data for kernel that is being profiled.
- The various statistics that are 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 at 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 workload distribution. This may result in some inaccuracy in the analysis in some cases.

<table>
<thead>
<tr>
<th>Show all columns</th>
<th>GPU Timestamp (us)</th>
<th>GPU Time (us)</th>
<th>dynamic shared memory per block (bytes)</th>
<th>static shared memory per block (bytes)</th>
<th>l1 local load hit TypeSM Run</th>
<th>l1 local load miss TypeSM Run</th>
<th>l1 global load TypeSM Run</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory Throughput Analysis</td>
<td>1 31720</td>
<td>5467.36</td>
<td>0</td>
<td>2056</td>
<td>0</td>
<td>0</td>
<td>519532</td>
</tr>
<tr>
<td>2 42978</td>
<td>5480.63</td>
<td>0</td>
<td>2056</td>
<td>0</td>
<td>0</td>
<td>516739</td>
<td></td>
</tr>
<tr>
<td>3 54214</td>
<td>5459.23</td>
<td>0</td>
<td>2056</td>
<td>0</td>
<td>0</td>
<td>520312</td>
<td></td>
</tr>
<tr>
<td>4 65564</td>
<td>5469.34</td>
<td>0</td>
<td>2056</td>
<td>0</td>
<td>0</td>
<td>520351</td>
<td></td>
</tr>
<tr>
<td>5 76334</td>
<td>5471.26</td>
<td>0</td>
<td>2056</td>
<td>0</td>
<td>0</td>
<td>520377</td>
<td></td>
</tr>
<tr>
<td>6 87428</td>
<td>5462.42</td>
<td>0</td>
<td>2056</td>
<td>0</td>
<td>0</td>
<td>518082</td>
<td></td>
</tr>
<tr>
<td>7 98280</td>
<td>5476.17</td>
<td>0</td>
<td>2056</td>
<td>0</td>
<td>0</td>
<td>516576</td>
<td></td>
</tr>
<tr>
<td>8 109750</td>
<td>5476.48</td>
<td>0</td>
<td>2056</td>
<td>0</td>
<td>0</td>
<td>511720</td>
<td></td>
</tr>
<tr>
<td>9 121316</td>
<td>5501.36</td>
<td>0</td>
<td>2056</td>
<td>0</td>
<td>0</td>
<td>510221</td>
<td></td>
</tr>
</tbody>
</table>
Jacobi Iteration: Kernel Profiling

**Limiting Factor**
- **Achieved Instruction Per Byte Ratio:** 1.05 (Balanced Instruction Per Byte Ratio: 3.50)
- **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)

**Hints:**
- 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 Analysis Table](image)
#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++ ) {
            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++;
}
#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;
}
## Performance

CPU: Intel Xeon X5680  
6 Cores @ 3.33GHz

GPU: NVIDIA Tesla M2070

<table>
<thead>
<tr>
<th>Execution</th>
<th>Time (s)</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU 6 OpenMP threads (v2)</td>
<td>39.7</td>
<td></td>
</tr>
<tr>
<td>CPU 6 OpenMP threads (v3)</td>
<td>20.4</td>
<td>1.95x</td>
</tr>
<tr>
<td>OpenACC GPU (v2)</td>
<td>11.0</td>
<td></td>
</tr>
<tr>
<td>OpenACC GPU (v3)</td>
<td>5.7</td>
<td>1.93x</td>
</tr>
</tbody>
</table>
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
Questions?