GPU Computing with OpenACC Directives

Dr. Timo Stich
Developer Technology Group
NVIDIA Corporation
WHAT IS GPU COMPUTING?
Add GPUs: Accelerate Science Applications

CPU + GPU
Small Changes, Big Speed-up

Application Code

Compute-Intensive Functions
Use GPU to Parallelize

Rest of Sequential CPU Code

GPU

CPU
3 Ways to Accelerate Applications

- Libraries
  - “Drop-in” Acceleration

- OpenACC Directives
  - Easily Accelerate Applications

- Programming Languages
  - Maximum Performance
3 Ways to Accelerate Applications

- Libraries: “Drop-in” Acceleration
- OpenACC Directives: Easily Accelerate Applications
- Programming Languages: Maximum Flexibility
GPU ARCHITECTURE
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
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 150 GB/s for Quadro and 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
1. Copy input data from CPU memory to GPU memory
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
CUDA PROGRAMMING MODEL
OpenACC and CUDA

- OpenACC on NVIDIA GPUs compiles to target the CUDA platform
- CUDA is a parallel computing platform and programming model invented by NVIDIA.
- There is also CUDA C, CUDA Fortran, OpenCL, ...
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>Executes functions</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td></td>
<td></td>
</tr>
<tr>
<td>GPU</td>
<td>Device</td>
<td>Executes kernels</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

```c
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
- Blocks are grouped into a grid
- A kernel is executed as a grid of blocks of threads
Each kernel is executed on one device.

Multiple kernels can execute on a device at one time.

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
Memory hierarchy

- **Thread:**
  - Registers
  - Local memory

- **Block of threads:**
  - Shared memory

- **All blocks:**
  - Global memory

![Diagram of memory hierarchy with NVIDIA logo](nvidia-memory-hierarchy-diagram)
OpenACC Directives

Your original Fortran or C code

Simple Compiler hints

Compiler Parallelizes code

Works on many-core GPUs & multicore CPUs

Program myscience

... serial code ...

!$acc kernels
do k = 1,n1
do i = 1,n2
... parallel code ...
enddo
enddo

!$acc end kernels

... End Program myscience

CPU

GPU

OpenACC Compiler Hint
Familiar to OpenMP Programmers

OpenMP

main() {
    double pi = 0.0; long i;

    #pragma omp parallel for reduction(+:pi)
    for (i=0; i<N; i++)
    {
        double t = (double)((i+0.05)/N);
        pi += 4.0/(1.0+t*t);
    }

    printf("pi = %f\n", pi/N);
}

OpenACC

main() {
    double pi = 0.0; long i;

    #pragma acc kernels
    for (i=0; i<N; i++)
    {
        double t = (double)((i+0.05)/N);
        pi += 4.0/(1.0+t*t);
    }

    printf("pi = %f\n", pi/N);
}
OpenACC
Open Programming Standard for Parallel Computing

“OpenACC will enable programmers to easily develop portable applications that maximize the performance and power efficiency benefits of the hybrid CPU/GPU architecture of Titan.”

--Buddy Bland, Titan Project Director, Oak Ridge National Lab

“OpenACC is a technically impressive initiative brought together by members of the OpenMP Working Group on Accelerators, as well as many others. We look forward to releasing a version of this proposal in the next release of OpenMP.”

--Michael Wong, CEO OpenMP Directives Board
Easy: Directives are the easy path to accelerate compute intensive applications

Open: OpenACC is an open GPU directives standard, making GPU programming straightforward and portable across parallel and multi-core processors

Powerful: GPU Directives allow complete access to the massive parallel power of a GPU
High-level, with low-level access

- Compiler directives to specify parallel regions in C, C++, Fortran
  - OpenACC compilers offload parallel regions from host to accelerator
  - Portable across OSes, host CPUs, accelerators, and compilers
- Create high-level heterogeneous programs
  - Without explicit accelerator initialization,
  - Without explicit data or program transfers between host and accelerator
- Programming model allows programmers to start simple
  - Enhance with additional guidance for compiler on loop mappings, data location, and other performance details
- Compatible with other GPU languages and libraries
  - Interoperate between CUDA C/Fortran and GPU libraries
  - e.g. CUFFT, CUBLAS, CUSPARSE, etc.
Focus on Exposing Parallelism

With Directives, tuning work focuses on *exposing parallelism*, which makes codes inherently better.

**Example:** Application tuning work using directives for new Titan system at ORNL

- **S3D**
  - Research more efficient combustion with next-generation fuels

- **CAM-SE**
  - Answer questions about specific climate change adaptation and mitigation scenarios

- Tuning top 3 kernels (90% of runtime)
- 3 to 6x faster on CPU+GPU vs. CPU+CPU
- But also improved all-CPU version by 50%

- Tuning top key kernel (50% of runtime)
- 6.5x faster on CPU+GPU vs. CPU+CPU
- Improved performance of CPU version by 100%
OpenACC Specification and Website

- Full OpenACC 1.0 Specification available online
- Quick reference card also available
- Beta implementations available now from PGI, Cray, and CAPS
Start Now with OpenACC Directives

Sign up for a free trial of the directives compiler now!

Free trial license to PGI Accelerator
Tools for quick ramp

www.nvidia.com/gpudirectives
A Very Simple Exercise: SAXPY

**SAXPY in C**

```c
void saxpy(int n, 
    float a, 
    float *x, 
    float *restrict y)
{
    #pragma acc kernels
    for (int i = 0; i < n; ++i)
        y[i] = a*x[i] + y[i];
}
...

// Perform SAXPY on 1M elements
saxpy(1<<20, 2.0, x, y);
...
```

**SAXPY in Fortran**

```fortran
subroutine saxpy(n, a, x, y)
    real :: x(:), y(:), a
    integer :: n, i

$!acc kernels
    do i=1,n
        y(i) = a*x(i)+y(i)
    enddo
$!acc end kernels
end subroutine saxpy

...

$ Perform SAXPY on 1M elements
call saxpy(2**20, 2.0, x_d, y_d)
...
```
Directive Syntax

- **Fortran**
  
  \[ \texttt{!$acc directive [clause [,] clause] ...} \]

  Often paired with a matching end directive surrounding a structured code block

  \[ \texttt{!$acc end directive} \]

- **C**
  
  \[ \texttt{#pragma acc directive [clause [,] clause] ...} \]

  Often followed by a structured code block
Each loop executed as a separate *kernel* on the GPU.

```
!$acc kernels
  do i=1,n
    a(i) = 0.0
    b(i) = 1.0
    c(i) = 2.0
  end do
  do i=1,n
    a(i) = b(i) + c(i)
  end do
!$acc end kernels
```

**Kernel:**
A parallel function that runs on the GPU
Kernels Construct

Fortran

```fortran
$acc kernels [clause ...]
    structured block
$acc end kernels
```

C

```c
#pragma acc kernels [clause ...]
    { structured block }
```

Clauses

- `if(condition)`
- `async(expression)`

Also, any data clause (more later)
C tip: the restrict keyword

- Declaration of intent given by the programmer to the compiler
  - Applied to a pointer, e.g.
    ```c
    float *restrict ptr
    ```
  - Meaning: “for the lifetime of ptr, only it or a value directly derived from it (such as ptr + 1) will be used to access the object to which it points”*

- Limits the effects of pointer aliasing
- OpenACC compilers often require restrict to determine independence
  - Otherwise the compiler can’t parallelize loops that access ptr
  - Note: if programmer violates the declaration, behavior is undefined

Complete SAXPY example code

- Trivial first example
  - Apply a loop directive
  - Learn compiler commands

```c
#include <stdlib.h>

void saxpy(int n, float a, float *restrict x, float *restrict y)
{
  #pragma acc kernels
  for (int i = 0; i < n; ++i)
    y[i] = a * x[i] + y[i];
}

int main(int argc, char **argv)
{
  int N = 1<<20; // 1 million floats
  if (argc > 1)
    N = atoi(argv[1]);
  float *x = (float*)malloc(N * sizeof(float));
  float *y = (float*)malloc(N * sizeof(float));
  for (int i = 0; i < N; ++i) {
    x[i] = 2.0f;
    y[i] = 1.0f;
  }
  saxpy(N, 3.0f, x, y);
  return 0;
}

*restrict: "I promise y does not alias x"
```
Complete SAXPY example code

- Trivial first example
  - Apply a loop directive
  - Learn compiler commands

```c
#include <stdlib.h>

void saxpy(int n,
            float a,
            float *restrict x,
            float *restrict y)
{
    #pragma acc kernels
    for (int i = 0; i < n; ++i)
        y[i] = a * x[i] + y[i];
}

int main(int argc, char **argv)
{
    int N = 1<<20; // 1 million floats
    if (argc > 1)
        N = atoi(argv[1]);
    float *x = (float*)malloc(N * sizeof(float));
    float *y = (float*)malloc(N * sizeof(float));
    for (int i = 0; i < N; ++i) {
        x[i] = 2.0f;
        y[i] = 1.0f;
    }
    saxpy(N, 3.0f, x, y);
    return 0;
}
```

*restrict: “I promise y does not alias x”
Compile and run

- **C:**
  ```
pgcc -acc -ta=nvidia -Minfo=accel -o saxpy_acc saxpy.c
  ```

- **Fortran:**
  ```
pgf90 -acc -ta=nvidia -Minfo=accel -o saxpy_acc saxpy.f90
  ```

- **Compiler output:**
  ```
pgcc -acc -Minfo=accel -ta=nvidia -o saxpy_acc saxpy.c
  saxpy:
    8, Generating copyin(x[:n-1])
    Generating copy(y[:n-1])
    Generating compute capability 1.0 binary
    Generating compute capability 2.0 binary
    9, Loop is parallelizable
    Accelerator kernel generated
    9, #pragma acc loop worker, vector(256) /* blockIdx.x threadIdx.x */
    CC 1.0 : 4 registers; 52 shared, 4 constant, 0 local memory bytes; 100% occupancy
    CC 2.0 : 8 registers; 4 shared, 64 constant, 0 local memory bytes; 100% occupancy
  ```
DATA MANAGEMENT
Data Transfer

For efficiency, decouple data movement and compute off-load
Data Construct

Fortran

```fortran
!$acc data [clause ...]
structured block
!$acc end data
```

C

```c
#pragma acc data [clause ...]
{ structured block }
```

General Clauses

```c
    if( condition )
    async( expression )
```

Manage data movement. Data regions may be nested.
**Data Clauses**

**copy ( list )**  Allocates memory on GPU and copies data from host to GPU when entering region and copies data to the host when exiting region.

**copyin ( list )**  Allocates memory on GPU and copies data from host to GPU when entering region.

**copyout ( list )**  Allocates memory on GPU and copies data to the host when exiting region.

**create ( list )**  Allocates memory on GPU but does not copy.

**present ( list )**  Data is already present on GPU from another containing data region.

and **present_or_copy[in|out]**, **present_or_create**, **deviceptr**.
Array Shaping

- Compiler sometimes cannot determine size of arrays
  - Must specify explicitly using data clauses and array “shape”

- C
  ```c
  #pragma acc data copyin(a[0:size-1]), copyout(b[s/4:3*s/4])
  ```

- Fortran
  ```fortran
  !$pragma acc data copyin(a(1:size)), copyout(b(s/4:3*s/4))
  ```

- Note: data clauses can be used on data, kernels or parallel
for(int j=0; j<k; ++j)
{
    saxpy( n, a, x, y );
}

void saxpy( int n, float a, float* x, float* restrict y )
{
    int i;
    #pragma acc kernels
    for( i=0; i<n; ++i )
    {
        y[i] += a*x[i];
    }
}
Data Region across Multiple Procedures

```c
void saxpy( int n, float a, float* x, float* restrict y )
{
    int i;
    #pragma acc kernels
    for( i=0; i<n; ++i )
        y[i] += a*x[i];
}
```

Data is copied from host to device

```c
for(int j=0; j<k; ++j)
{
    saxpy( n, a, x, y );
}
```

Data is copied from device to host
Data Region across Multiple Procedures

```c
#pragma acc data \
copy(x[0:n],y[0:n])
for(int j=0; j<k; ++j)
{
    saxpy(n, a, x, y);
}
```
Data Region across Multiple Procedures

```c
#pragma acc data (copy(x[0:n],y[0:n]))
for(int j=0; j<k; ++j)
{
    saxpy( n, a, x, y );
}
```

```c
void saxpy( int n, float a, float* x, float* restrict y )
{
    int i;
    #pragma acc kernels
    for( i=0; i<n; ++i )
        y[i] += a*x[i];
}
```
Update Construct

Fortran

!$acc update [clause ...]

C

#pragma acc update [clause ...]

Clauses

host( list )
device( list )

if( expression )
async( expression )

Used to update existing data after it has changed in its corresponding copy (e.g. update device copy after host copy changes)

Move data from GPU to host, or host to GPU. Data movement can be conditional, and asynchronous.
KERNELS, PARALLEL, LOOP
Kernels Construct (Recap)

Fortran

```fortran
!$acc kernels [clause ...]
   structured block
!$acc end kernels
```

C

```c
#include <openacc.h>

#pragma acc kernels [clause ...]
   { structured block }
```

Clauses

- `if( condition )`
- `async( expression )`

Also, any data clause (more later)
Parallel Construct

Fortran

!$acc parallel [clause …]
  structured block
!$acc end parallel

C

#pragma acc parallel [clause …]
  { structured block }

Clauses

if( condition )
async( expression )

Also, any data clause (more later)
Parallel Construct

Alternative to kernels directive
- Behavior matches OpenMP parallel directive

Explicitly start a gang of workers to execute in parallel
- Use loop construct to actually split the work!

PGI insider Article
- [http://www.pgroup.com/lit/articles/insider/v4n2a1.htm](http://www.pgroup.com/lit/articles/insider/v4n2a1.htm)
Advanced OpenACC

(Afternoon Session)
GANGS, WORKERS, VECTOR
The OpenACC execution model has three levels: 
- *gang*, *worker* and *vector*

Maps to Virtual Architecture of Processing Elements (PEs)
- Each PE is multithreaded and each thread can execute vector instructions

For GPUs one possible mapping could be *gang*=block, *worker*=warp, *vector*=threads in a warp
- Note: Can vary with compiler vendor
Mapping OpenACC to CUDA threads and blocks

```c
#pragma acc kernels
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];
}```
Mapping OpenACC to CUDA threads and blocks

```c
#pragma acc parallel 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.

Compiler can notice that only 'gangs' are being created, so it might decide to create threads instead, say 2 thread blocks of 50 threads.
Mapping OpenACC to CUDA threads and blocks

```c
#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

```c
#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 multi dimensional blocks and grids to OpenACC

A nested for loop generates multidimensional blocks and grids

```c
#pragma acc kernels loop gang(100), vector(16)
  for( ... )
#pragma acc loop gang(200), vector(32)
  for( ... )
```

The block is 100 blocks tall (row/Y direction) and 32 thread wide (column/X direction).
Selecting block size (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 the number of 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...
Selecting grid size (number of gangs)

- Most obvious mapping is to have # of gangs times # of workers times # of vectors equal the total problem size
  - Threads can be assigned to 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?
  - At least enough threads to fill the GPU several times over
  - Typically 100,000+
Warps - Vectors of Threads

- **Group of 32 Threads**
  - Share one Program Counter (PC)
  - Threads of warp execute in lock-step (similar to Vectorprocessor)
- **Divergence within warps is supported on the HW level**
  - Improves maintainability and flexibility over explicit vector programming
  - Automatic serialization/instruction replays

<table>
<thead>
<tr>
<th>Warp</th>
<th>ThreadIdx</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0 1 ... 30 31</td>
</tr>
<tr>
<td>1</td>
<td>32 33 ... 62 63</td>
</tr>
</tbody>
</table>
Divergence Examples - No Serialization

// ...
if ( threadIdx.x < 32 ) {
    // Do something
} else {
    // Do something else
}
// ...

<table>
<thead>
<tr>
<th>Warp 0</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>3</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Warp 1</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>3</td>
<td>3</td>
<td>6</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>3</td>
<td>3</td>
</tr>
</tbody>
</table>
Divergence Examples - No Serialization

// ...
if( threadIdx.x < 32 ) {
    // Do something
} else {
    // Do something else
}
// ...

Warp 0
0 0
0 1

Warp 1
3 3
2 3
3 6

0 3 1
3 2 3

6 3
Divergence Examples - No Serialization

if ( threadIdx.x < 32 ) {
    // Do something
} else {
    // Do something else
}

// ...

<table>
<thead>
<tr>
<th>0</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>3</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>3</td>
</tr>
</tbody>
</table>
if( threadIdx.x < 16 ) {
    // Do something
} else {
    // Do something else
}
Divergence Examples - Serialization

// ...
if ( threadIdx.x < 16 ) {
    // Do something
} else {
    // Do something else
}
// ...

Warp 0

Warp 1
Divergence Examples - Serialization

// ...
if( threadIdx.x < 16 ) {
    // Do something
} else {
    // Do something else
}
// ...
Divergence Examples - Serialization

// ...
if(threadIdx.x < 16)
{
    // Do something
}
else
{
    // Do something else
}
// ...
Fermi Memory Hierarchy Review

- **Local storage**
  - Each thread has own local storage
  - Mostly registers (managed by the compiler)

- **Shared memory / L1**
  - Program configurable: 16KB shared / 48 KB L1  OR  48KB shared / 16KB L1
  - Shared memory is accessible by the threads in the same threadblock
  - Very low latency
  - Very high throughput: 1+ TB/s aggregate

- **L2**
  - All accesses to global memory go through L2, including copies to/from CPU host

- **Global memory**
  - Accessible by all threads as well as host (CPU)
  - High latency (400-800 cycles)
  - Throughput: up to 177 GB/s
Fermi GMEM Operations

Two types of loads:

- **Caching**
  - Default mode
  - Attempts to hit in L1, then L2, then GMEM
  - Load granularity is **128-byte line**

- **Non-caching**
  - Compile with `-Xptxas -dlcm=cg` option to nvcc
  - Attempts to hit in L2, then GMEM
    - Do not hit in L1, invalidate the line if it’s in L1 already
  - Load granularity is **32-bytes**

Stores:

- Invalidate L1, write-back for L2
Load Operation

Memory operations are issued per warp (32 threads)

- Just like all other instructions
- Prior to Fermi, memory issues were per half-warp

Operation:

- Threads in a warp provide memory addresses
- Determine which lines/segments are needed
- Request the needed lines/segments
Warp requests 32 aligned, consecutive 4-byte words
Addresses fall within 1 cache-line
- Warp needs 128 bytes
- 128 bytes move across the bus on a miss
- Bus utilization: 100%

addresses from a warp

Memory addresses
Non-caching Load

- Warp requests 32 aligned, consecutive 4-byte words
- Addresses fall within 4 segments
  - Warp needs 128 bytes
  - 128 bytes move across the bus on a miss
  - Bus utilization: 100%

```
addresses from a warp
```

```
Memory addresses
```

```
0 32 64 96 128 160 192 224 256 288 320 352 384 416 448
```
Warp requests 32 aligned, permuted 4-byte words

Addresses fall within 1 cache-line
- Warp needs 128 bytes
- 128 bytes move across the bus on a miss
- Bus utilization: 100%
Non-caching Load

- Warp requests 32 aligned, permuted 4-byte words
- Addresses fall within 4 segments
  - Warp needs 128 bytes
  - 128 bytes move across the bus on a miss
  - Bus utilization: **100%**

Addresses from a warp

Memory addresses

0  32  64  96  128  160  192  224  256  288  320  352  384  416  448
Caching Load

- Warp requests 32 misaligned, consecutive 4-byte words
- Addresses fall within 2 cache-lines
  - Warp needs 128 bytes
  - 256 bytes move across the bus on misses
  - Bus utilization: 50%

Addresses from a warp

Memory addresses

0  32  64  96  128  160  192  224  256  288  320  352  384  416  448
Non-caching Load

- Warp requests 32 misaligned, consecutive 4-byte words
- Addresses fall within at most 5 segments
  - Warp needs 128 bytes
  - 256 bytes move across the bus on misses
  - Bus utilization: at least 80%
    - Some misaligned patterns will fall within 4 segments, so 100% utilization

addresses from a warp

Memory addresses
Caching Load

- All threads in a warp request the same 4-byte word
- Addresses fall within a single cache-line
  - Warp needs 4 bytes
  - 128 bytes move across the bus on a miss
  - Bus utilization: **3.125%**
Non-caching Load

- All threads in a warp request the same 4-byte word
- Addresses fall within a single segment
  - Warp needs 4 bytes
  - 32 bytes move across the bus on a miss
  - Bus utilization: 12.5%

Addresses from a warp

Memory addresses

0 32 64 96 128 160 192 224 256 288 320 352 384 416 448
Caching Load

- Warp requests 32 scattered 4-byte words
- Addresses fall within $N$ cache-lines
  - Warp needs 128 bytes
  - $N \times 128$ bytes move across the bus on a miss
  - Bus utilization: $\frac{128}{N \times 128}$

Memory addresses

addresses from a warp
Non-caching Load

- Warp requests 32 scattered 4-byte words
- Addresses fall within $N$ segments
  - Warp needs 128 bytes
  - $N\times32$ bytes move across the bus on a miss
  - Bus utilization: $\frac{128}{N\times32}$
Impact of Address Alignment

- Warps should access aligned regions for maximum memory throughput
  - Fermi L1 can help for misaligned loads if several warps are accessing a contiguous region
  - ECC further significantly reduces misaligned store throughput

Experiment:
- Copy 16MB of floats
- 256 threads/block

Greatest throughput drop:
- GT200: 40%
- Fermi:
  - CA loads: 15%
  - CG loads: 32%
Data Layout

Array of Structs (AOS)

```c
struct myStruct_t {
  float a;
  float b;
  int c;
}
myStruct_t myData[];
```

Struct of Arrays (SOA)

```c
struct {
  float a[];
  float b[];
  int c[];
} myData;
```

```c
#pragma acc kernel
for(int i=0;i<n;++i) {
  ... myData[i].a ...
}
```
#pragma acc kernel
for(int i=0;i<n;++i) {
    ... myData[i].a ... 
}
#pragma acc kernel
for(int i=0;i<n;++i) {
    ... myData[i].a ...
}
## Data Layout

### Address Layout

<table>
<thead>
<tr>
<th>Address</th>
<th>0</th>
<th>4</th>
<th>8</th>
<th>12</th>
<th>16</th>
<th>20</th>
<th>24</th>
<th>28</th>
<th>32</th>
</tr>
</thead>
</table>

```c
#pragma acc kernel
for(int i=0; i<n; ++i) {
    ... myData[i].a ... 
}
```
Cache Construct

Fortran

```fortran
!$acc cache [clause ...]
    structured block
!$acc end cache
```

C

```c
#pragma acc cache [clause ...]
    { structured block }
```

- Specifies array elements or subarrays that should be fetched into the highest level of the cache for the body of the loop.
Shared Memory

**Uses:**
- Inter-thread communication within a block
- Cache data to reduce redundant global memory accesses
- Use it to improve global memory access patterns

**Organization:**
- 32 banks, 4-byte wide banks
- Successive 4-byte words belong to different banks

**Performance:**
- Very high throughput: 1+ TB/s aggregate
void saxpy(int n,
    float a,
    float *restrict *x,
    float *restrict y)
{
    #pragma acc kernels
    for (int i = 0; i < n; ++i) {
        y[i] = a * x[i] + y[i];
    }
}
#pragma acc kernels
for (int i = 0; i < n; ++i) {
    saxpy(...);
}

Currently inlining is mandatory

Compiler flag: -Minline[=levels:3]
Async Clause

**Synchronous Operation**
- Control does not return until accelerator operation has finished

**Asynchronous Operation**
- Control returns immediately
- Explicit synchronization with `acc sync`

**Allows for both CPU and GPU to work in parallel**
- Heterogeneous computing at its best
Async Clause

```c
#pragma acc kernels loop async
for( int i=0; i < N; ++i ) {
    ...
}
#pragma acc update host \ (x[0:N]) async

cpu_workload()

#pragma acc wait
```
Async Clause

#pragma acc kernels loop async
for( int i=0; i < N; ++i ) {
    ...
}
#pragma acc update host \ 
    (x[0:N]) async

cpu_workload()

#pragma acc wait

With async control returns immediately to CPU

CPU sync with GPU
Multiple GPUs

Assign one device to each CPU thread
- One GPU can be the accelerator for multiple CPU threads

Device Management API

- `acc_get_num_devices(acc_device_nvidia);`
- `acc_set_device_num(i, acc_device_nvidia);`
OpenACC and MPI

#include <openacc.h>
#include <mpi.h>

int myrank;
MPI_Comm_rank( MPI_COMM_WORLD, &myrank );

int numdev = acc_get_num_devices( acc_device_nvidia );
int i = myrank % numdev;
acc_set_device_num( i, acc_device_nvidia );
OpenACC and OpenMP

```c
#include <openacc.h>
#include <omp.h>

#pragma omp parallel num_threads(12)
{
    int numdev = acc_get_num_devices( acc_device_nvidia );
    int i = omp_get_threadnum() % numdev;
    acc_set_device_num( i, acc_device_nvidia );
}
```
Using CUDA Libraries with OpenACC
3 Ways to Accelerate Applications

Applications

Libraries

OpenACC Directives

Programming Languages

“Drop-in” Acceleration

Easily Accelerate Applications

CUDA Libraries are interoperable with OpenACC

Maximum Flexibility
3 Ways to Accelerate Applications

Applications

Libraries

“Drop-in” Acceleration

OpenACC Directives

Easily Accelerate Applications

Programming Languages

CUDA Languages are interoperable with OpenACC, too!
GPU Accelerated Libraries
“Drop-in” Acceleration for Your Applications
CUDA Math Libraries

High performance math routines for your applications:
- cuFFT - Fast Fourier Transforms Library
- cuBLAS - Complete BLAS Library
- cuSPARSE - Sparse Matrix Library
- cuRAND - Random Number Generation (RNG) Library
- NPP - Performance Primitives for Image & Video Processing
- Thrust - Templated C++ Parallel Algorithms & Data Structures
- math.h - C99 floating-point Library

Included in the CUDA Toolkit   Free download @ www.nvidia.com/getcuda

More information on CUDA libraries:
http://www.nvidia.com/object/gtc2010-presentation-archive.html#session2216
Sharing data with libraries

- CUDA libraries and OpenACC both operate on device arrays

- OpenACC provides mechanisms for interop with library calls
  - `deviceptr data clause`
  - `host_data construct`

- Note: same mechanisms useful for interop with custom CUDA C/C++/Fortran code
deviceptr Data Clause

deviceptr( list ) Declares that the pointers in list refer to device pointers that need not be allocated or moved between the host and device for this pointer.

Example:

C
#pragma acc data deviceptr(d_input)

Fortran
$!acc data deviceptr(d_input)
host_data Construct

Makes the address of device data available on the host.

deviceptr( list ) Tells the compiler to use the device address for any variable in list. Variables in the list must be present in device memory due to data regions that contain this construct.

Example

C

#pragma acc host_data use_device(d_input)

Fortran

$!acc host_data use_device(d_input)
Thank you