OpenACC Fundamentals
OpenMP and/versus OpenACC

• OpenMP started as CPU-only specification that now includes accelerators
  - Continues to have support from major CPU vendors
  - Now includes advanced support GPUs and other accelerators

• OpenACC unified many offloading solutions for accelerators and now includes CPU as a target
  - It may be used with GNU (first stable in version 8.1.0) and PGI compilers
  - US' DOE sponsors further development
    • This may introduce LLVM support
  - Support for CPUs was added along the way as one of OpenACC targets
Programming Hardware Accelerators

Scientific Applications

Software Libraries

Compiler directives

Programming Languages

Drop-in Replacement of Relevant Functions

Easy Acceleration of Custom Code

- Flexibility of implementation
- Steep learning curve
- Maximum performance
- Error Prone
OpenACC Directives

Manage Data Movement

```
#pragma acc data copyin(x,y) copyout(z)
{
    /* … */
}
```

Initiate Parallel Execution

```
#pragma acc parallel
{
    #pragma acc loop gang vector
    for (i = 0; i < n; ++i) {
        z[i] = x[i] + y[i];
        /* … */
    }
}
/* … */
```

Optimize Loop Mappings

OpenACC Directives for Accelerators

- Incremental
- Single source
- Interoperable
- Performance portable
- CPU, GPU, Xeon Phi
GPU Hardware Deep Dive
Processor and Accelerator

- **CPU**
  - Optimized for sequential tasks
  - Latency-bound computation
  - High context switching cost

- **Accelerator**
  - Optimized for parallel tasks
  - Data parallel computation
  - Low thread switching cost
Accelerated Computing: Software Perspective

CPU code

GPU code

Accelerator
## Comparison of More Recent NVIDIA GPUs

<table>
<thead>
<tr>
<th>Tesla cards</th>
<th>M40</th>
<th>K40</th>
<th>K80</th>
<th>P100</th>
<th>V100</th>
</tr>
</thead>
<tbody>
<tr>
<td>GPU</td>
<td>GM200</td>
<td>GK110</td>
<td>2x GK210</td>
<td>GP100</td>
<td>GV100</td>
</tr>
<tr>
<td>Name</td>
<td>Maxwell</td>
<td>Kepler</td>
<td>Kepler</td>
<td>Pascal</td>
<td>Volta</td>
</tr>
<tr>
<td>SMs</td>
<td>24</td>
<td>15</td>
<td>13+13</td>
<td>56 (28 TPCs)</td>
<td>80 (40 TPCs)</td>
</tr>
<tr>
<td>CUDA cores</td>
<td>3072 (96 FP64 cores)</td>
<td>2880 (960 FP64 cores)</td>
<td>2x 2496</td>
<td>3584 (1792 FP64)</td>
<td>5120 (640 tensor cores)</td>
</tr>
<tr>
<td>Base Clock MHz</td>
<td>948</td>
<td>745</td>
<td>560</td>
<td>1328</td>
<td></td>
</tr>
<tr>
<td>Boost Clock MHz</td>
<td>1114</td>
<td>810/875</td>
<td>875</td>
<td>1480</td>
<td>1455</td>
</tr>
<tr>
<td>Peak FP64 Tflop/s</td>
<td>0.213</td>
<td>1.680</td>
<td>2.91</td>
<td>5.304</td>
<td>7.5</td>
</tr>
<tr>
<td>Peak FP32 Tflop/s</td>
<td>7</td>
<td>5.04</td>
<td>8.73</td>
<td>10.6</td>
<td>15 (120 Tensor Core)</td>
</tr>
<tr>
<td>Texture Units</td>
<td>192</td>
<td>240</td>
<td>208+208</td>
<td>224</td>
<td>320</td>
</tr>
<tr>
<td>Register File KB</td>
<td>6144</td>
<td>3840</td>
<td>3328+3328</td>
<td>14336</td>
<td>20480</td>
</tr>
<tr>
<td>L2 Cache Size KB</td>
<td>3072</td>
<td>1536</td>
<td>3072</td>
<td>4096</td>
<td>6144</td>
</tr>
<tr>
<td>Memory type</td>
<td>GDDR5</td>
<td>GDDR5</td>
<td>GDDR5</td>
<td>HBM2</td>
<td>HBM2</td>
</tr>
<tr>
<td>Memory interface bits</td>
<td>384</td>
<td>384</td>
<td>384</td>
<td>4096</td>
<td>4096</td>
</tr>
<tr>
<td>Memory size GB</td>
<td>24</td>
<td>12</td>
<td>24 (12+12)</td>
<td>16</td>
<td>16 GB</td>
</tr>
<tr>
<td>Peak bandwidth GB/s</td>
<td>288</td>
<td>288</td>
<td>480 (240+240)</td>
<td>720</td>
<td>900</td>
</tr>
<tr>
<td>TDP W</td>
<td>250</td>
<td>235</td>
<td>300</td>
<td>300</td>
<td>300</td>
</tr>
<tr>
<td>Transistors billions</td>
<td>8</td>
<td>7.1</td>
<td>7.1+7.1</td>
<td>15.3</td>
<td>21.2</td>
</tr>
<tr>
<td>Die size mm²</td>
<td>601</td>
<td>551</td>
<td>561+561</td>
<td>610</td>
<td>815</td>
</tr>
<tr>
<td>Manufacturing process</td>
<td>28 nm</td>
<td>28 nm</td>
<td>28 nm</td>
<td>16 nm</td>
<td>12 nm</td>
</tr>
</tbody>
</table>
GPU Device Structure

- Global memory
  - Analogous to RAM in a CPU server
  - Accessible by both GPU and CPU
  - Currently up to 16 GB in Tesla products
- Streaming Multiprocessors (SM)
  - Performs the actual computation
  - Each SM has its own: Control units, registers, execution pipelines, caches
- SM has many CUDA Cores per SM: architecture dependent
- SM has Special-function units: exp/cos/sin/tan, etc.
- SM has Shared memory + L1 cache
- SM has thousands of 32-bit registers
- CUDA core has floating point (IEEE 754-2008, FMA) & Integer unit; logic, move, compare, branch units
GPU Execution Model: Abstraction above Device

Hardware
Scalar Processor:

Multiprocessor:

Device

Software

Threads are executed by scalar processors

Thread blocks are executed on multiprocessor

Thread blocks do not migrate

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

A kernel is launched as a grid of thread blocks
GPU Warp: Basic Unit of Hardware Execution

- A thread block consists of 32-thread warps
- A warp is executed physically in parallel (SIMT) on a multiprocessor
Memory System on each SM

- Extremely fast, but small, i.e., 10s of kB
- Programmer chooses whether to use cache as L1 or Shared Memory
- L1
  - Hardware-managed—used for things like register spilling
  - Should NOT attempt to utilize like CPU caches
- Shared Memory—programmer MUST synchronize data accesses!!!
  - User-managed scratch pad
  - Repeated access to same data or multiple threads with same data
Low-Latency, Throughput-Oriented Processing

- CPU architecture minimizes latency within each thread with:
  - Cache hierarchy: L1, L2, L3, L4
  - Data prefetcher
  - Out-of-order scheduler
  - Branch predictor
  - Register renaming
  - ...
- Context switch is expensive
  - Especially for kernel-managed threads

- GPU architecture hides latency with computation from other thread warps
- GPU scheduler can switch between warps in a few cycles because all resources are already allocated on SM
Memory Coalescing for Reads and Writes

- Global memory access happens in transactions of 32 or 128 bytes only
- The hardware will try to reduce requests to as few transactions as possible
- Coalesced access:
  - A group of 32 contiguous threads (“warp”) accessing adjacent words in global memory
  - Fewer transactions and high utilization of GDDR5/HBM2 bandwidth
- Un-coalesced access:
  - A warp of 32 threads accessing scattered words
  - Many transactions and low utilization of bandwidth
- Optimization tip:
  - If the data must be scattered, try to gather the data into shared memory explicitly and load/store from there

Coalesced memory access

![Coalesced memory access diagram](image)

Single transaction of 128 bytes = 32 x 4 bytes

Un-coalesced memory access

![Un-coalesced memory access diagram](image)

32 bytes in transaction but only 4 bytes used: 4 / 32 = 12.5% bandwidth used
CPU SIMD (SSE, AVX, AltiVec, NEON) vs GPU SIMT

Single Instruction Multiple Data (SIMD)
- Vector instructions perform the same operation on multiple data elements.
- Data must be loaded and stored in contiguous buffers
- Either the programmer or the compiler must generate vector instructions

Single Instruction Multiple Thread (SIMT)
- Scalar instructions execute simultaneously by multiple hardware threads
- Contiguous data not required.
- So if something can run in SIMD, it can run in SIMT, but not necessarily the reverse.
- SIMT can better handle indirection
- The hardware enables parallel execution of scalar instructions
SIMD, SIMT Branching and Converged Execution

**SIMD**
- Execute converged instructions
- Generate vector mask for true
- Execute masked vector instruction
- Generate vector mask for false
- Execute masked vector instruction
- Continue to execute converged instructions
- Divergence (hopefully) handled by compiler through masks and/or gather/scatter operations.

**SIMT**
- Execute converged instructions
- Executed true branch
- Execute false branch
- Continue to execute converged instructions
- Divergence handle by hardware through predicated instructions.