INTRODUCTION TO GPU ARCHITECTURE AND GPU COMPUTING

BRENT LEBACK, MEMBER OF THE NVIDIA HPC SDK TEAM
**NVIDIA HPC SDK**

Available at developer.nvidia.com/hpc-sdk, on NGC, via Spack, and in the Cloud

---

### DEVELOPMENT

<table>
<thead>
<tr>
<th>Programming Models</th>
<th>Compilers</th>
<th>Core Libraries</th>
<th>Math Libraries</th>
<th>Communication Libraries</th>
</tr>
</thead>
<tbody>
<tr>
<td>Standard C++ &amp; Fortran</td>
<td>nvcc</td>
<td>libcu++</td>
<td>cuBLAS</td>
<td>HPC-X</td>
</tr>
<tr>
<td>OpenACC &amp; OpenMP</td>
<td>nvc++</td>
<td>Thrust</td>
<td>cuTENSOR</td>
<td>MPI</td>
</tr>
<tr>
<td>CUDA</td>
<td>nvfortran</td>
<td>CUB</td>
<td>cuSPARSE</td>
<td>UCX</td>
</tr>
</tbody>
</table>

### ANALYSIS

<table>
<thead>
<tr>
<th>Profilers</th>
<th>Debugger</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nsight</td>
<td>cuda-gdb</td>
</tr>
<tr>
<td>Systems</td>
<td>Host</td>
</tr>
<tr>
<td>Compute</td>
<td>Device</td>
</tr>
</tbody>
</table>

---

Develop for the NVIDIA Platform: GPU, CPU and Interconnect

Libraries | Accelerated C++ and Fortran | Directives | CUDA

7-8 Releases Per Year | Freely Available
A SLIDE FROM A TALK FIVE YEARS AGO

XXX CPU

Tesla Accelerator

High Capacity Memory

High Bandwidth Memory

Shared Cache

Shared Cache

NVLink
PROCESSOR COUNTS THROUGH THE YEARS

- Parallelism via MPI
- Performance improvements via manufacturing/process improvements, ILP, more registers, faster clocks.
- CPU SW gained features like dynamic memory allocation, large heaps, large stack.
THE ADVENT OF SIMD HARDWARE AND INSTRUCTIONS

- High-level Parallelism still via MPI
- Performance improvements via manufacturing/process improvements, ILP, more registers, faster clocks.
- Vectorization of loops becomes important to take advantage of SIMD lanes
- Some programmers resort to SIMD intrinsics
- Still code to the main core/sequencer.
MULTICORE ARCHITECTURE WITH SIMD INSTRUCTIONS

- High-level Parallelism via MPI and perhaps OpenMP
- Clock rates begin to slow
- NUMA issues begin. libnuma/pthreads makes its way into the linux kernel
- Memory bandwidth does not keep up with compute speed
- Main memory has to be shared among the cores
- More and bigger caches
MANYCORE WITH AVX-512

- High-level Parallelism via MPI and perhaps OpenMP
- AVX-512 HW slow to take hold, initial implementations not optimal
- NUMA issues continue.
- Vectorizing compilers are still important
- SIMD intrinsics still in use
- Memory bandwidth does not keep up with compute speed
- Main memory has to be shared among the cores
- More and bigger caches
- SW grows in complexity, relies on features like dynamic memory allocation, large heaps, large data/call stacks.
- Still code to the main core/sequencer.
AMPERE A100 AND AVX-512 CPU
AMPERE HW CHARACTERISTICS

- High-level Parallelism via MPI and perhaps OpenMP
- Multiple CUDA contexts, CUDA streams, MIG, MPS allow sharing the GPU resource
- Synchronization between columns in the diagram is “hard”
- Offloading compilers are important, kernel scheduling, tuning and flexibility of launch parameters is key.

- Memory bandwidth is many times higher than a CPU
- Memory latency is high, caches are relatively small
- Programmer-managed shared memory (cache) is useful for performance and to communicate between cores in an SM
- Massively oversubscribing the cores is a key to performance

- Code to the core/lane. OS/low-level runtime handles divergence, at a cost
- Each core/lane loads and stores its own data. OS/low-level runtime ideally coalesces those into contiguous blocks

- Each core/lane has a small stack, limited number of registers, compared to a CPU core.
- Overheads can adversely affect performance since each core/lane only targets a small number of array elements
1.2. CUDA Best Practices

The performance guidelines and best practices described in the CUDA C++ Programming Guide and the CUDA C++ Best Practices Guide apply to all CUDA-capable GPU architectures. Programmers must primarily focus on following those recommendations to achieve the best performance.

The high-priority recommendations from those guides are as follows:

- Find ways to parallelize sequential code.
- Minimize data transfers between the host and the device.
- Adjust kernel launch configuration to maximize device utilization.
- Ensure global memory accesses are coalesced.
- Minimize redundant accesses to global memory whenever possible.
- Avoid long sequences of diverged execution by threads within the same warp.
PROGRAMMING THE NVIDIA PLATFORM
CPU, GPU, and Network

ACCELERATED STANDARD LANGUAGES
ISO C++, ISO Fortran

std::transform(par, x, x+n, y, y,
    [=](float x, float y){ return y + a*x; })

do concurrent (i = 1:n)
y(i) = y(i) + a*x(i)
enddo

import cunumeric as np

def saxpy(a, x, y):
y[:] += a*x

#pragma acc data copy(x,y) {
    ...
    std::transform(par, x, x+n, y, y,
        [=](float x, float y){ return y + a*x; })
    ...
}

#pragma omp target data map(x,y) {
    ...
    std::transform(par, x, x+n, y, y,
        [=](float x, float y){ return y + a*x; })
    ...
}

_INCREMENTAL PORTABLE OPTIMIZATION
OpenACC, OpenMP

_PLATFORM SPECIALIZATION
CUDA

__global__ void saxpy(int n, float a,
    float *x, float *y) {
    int i = blockIdx.x*blockDim.x + threadIdx.x;
    if (i < n) y[i] += a*x[i];
}

int main(void) {
    ...
    cudaMemcpy(d_x, x, ...);
    cudaMemcpy(d_y, y, ...);
    saxpy<<<(N+255)/256,256>>>(...);
    cudaMemcpy(y, d_y, ...);

ACCELERATION LIBRARIES
Core  Math  Communication  Data Analytics  AI  Quantum