Parallel Accelerators

Přemysl Šůcha

``Parallel algorithms'', 2017/2018
CTU/FEL
Topic Overview

• Graphical Processing Units (GPU) and CUDA
• Vector addition on CUDA
• Intel Xeon Phi
• Matrix equations on Xeon Phi
Graphical Processing Units
GPU – Nvidia - Roadmap
GPU - Use

• GPU is especially well-suited to address problems that can be expressed as data-parallel computations.
• The same program is executed on many data elements in parallel - with high arithmetic intensity.
• Applications that process large data sets can use a data-parallel programming model to speed up the computations (3D rendering, image processing, video encoding, …)
• Many algorithms outside the field of image rendering and processing are accelerated by data-parallel processing too (machine learning, general signal processing, physics simulation, finance, …).
GPU - Overview

- CPU code runs on the **host**, GPU code runs on the **device**.
- A **kernel** consists of multiple threads.
- Threads execute in 32-thread groups called **warps**.
- Threads are grouped into **blocks**.
- A collection of blocks is called a **grid**.
GPU - Hardware Organization Overview

- GPU chip consists of one or more streaming multiprocessors (SM).
- A multiprocessor consists of 1 (CC 1.x), 2 (CC 2.x), or 4 (CC 3.x, 5.x, 6.x, 7.x) warp schedulers. (CC = CUDA Capability)
- Each warp scheduler can issue to 2 (CC 5 and 6) or 1 (CC 7) dispatch units.
- A multiprocessor consists of functional units of several types.
Streaming Multiprocessor (SM) - Volta
GPU - Functional Units

• **INT, FP32 (CUDA Core)** - functional units that executes most types of instructions, including most integer and single precision floating point instructions.

• **FP64 (Double Precision)** - executes double-precision floating point instructions.

• **SFU (Special Functional Unit)** - executes reciprocal and transcendental instructions such as sine, cosine, and reciprocal square root.

• **LD/ST (Load/Store Unit)** – handles load and store instructions.

• **TENSOR CORE** – for deep learning matrix arithmetic.
GPU - Tensor Core

- V100 GPU contains 640 Tensor Cores: eight (8) per SM.
- Tensor Core performs 64 floating point FMA (fused multiply–add) operations per clock.
- Matrix-Matrix multiplication (GEMM) operations are at the core of neural network training and inferencing.
- Each Tensor Core operates on a 4x4 matrices.

\[ D = \begin{pmatrix} A_{0,0} & A_{0,1} & A_{0,2} & A_{0,3} \\ A_{1,0} & A_{1,1} & A_{1,2} & A_{1,3} \\ A_{2,0} & A_{2,1} & A_{2,2} & A_{2,3} \\ A_{3,0} & A_{3,1} & A_{3,2} & A_{3,3} \end{pmatrix} \begin{pmatrix} B_{0,0} & B_{0,1} & B_{0,2} & B_{0,3} \\ B_{1,0} & B_{1,1} & B_{1,2} & B_{1,3} \\ B_{2,0} & B_{2,1} & B_{2,2} & B_{2,3} \\ B_{3,0} & B_{3,1} & B_{3,2} & B_{3,3} \end{pmatrix} + \begin{pmatrix} C_{0,0} & C_{0,1} & C_{0,2} & C_{0,3} \\ C_{1,0} & C_{1,1} & C_{1,2} & C_{1,3} \\ C_{2,0} & C_{2,1} & C_{2,2} & C_{2,3} \\ C_{3,0} & C_{3,1} & C_{3,2} & C_{3,3} \end{pmatrix} \]
Streaming Multiprocessor (SM)

- Each SM has a set of temporary registers split amongst threads.
- Instructions can access high-speed shared memory.
- Instructions can access a cache-backed constant space.
- Instructions can access local memory.
- Instructions can access global space. (very slow in general)
GPU Architecture - Volta
## GPU Architectures

<table>
<thead>
<tr>
<th>Tesla Product</th>
<th><strong>Tesla K40</strong></th>
<th><strong>Tesla M40</strong></th>
<th><strong>Tesla P100</strong></th>
<th><strong>Tesla V100</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Manufacturing Process</td>
<td>28nm</td>
<td>28nm</td>
<td>16nm</td>
<td>12nm</td>
</tr>
<tr>
<td>Transistors</td>
<td>7.1 Billion</td>
<td>8.0 Billion</td>
<td>15.3 Billion</td>
<td>21.1 Billion</td>
</tr>
<tr>
<td>SMs / GPU</td>
<td>15</td>
<td>24</td>
<td>28</td>
<td>40</td>
</tr>
<tr>
<td>F32 Cores / SM</td>
<td>192</td>
<td>128</td>
<td>64</td>
<td>64</td>
</tr>
<tr>
<td>F32 Cores / GPU</td>
<td>2880</td>
<td>3072</td>
<td>3584</td>
<td>5120</td>
</tr>
<tr>
<td>Peak FP32 TFLOPS</td>
<td>5</td>
<td>6.8</td>
<td>10.6</td>
<td>15.7</td>
</tr>
<tr>
<td>Peak FP64 TFLOPS</td>
<td>1.7</td>
<td>0.21</td>
<td>5.3</td>
<td>7.8</td>
</tr>
<tr>
<td>GPU Boost Clock</td>
<td>810/870 MHz</td>
<td>1114 MHz</td>
<td>1480 MHz</td>
<td>1530 MHz</td>
</tr>
<tr>
<td>TDP</td>
<td>235 W</td>
<td>250 W</td>
<td>300 W</td>
<td>300 W</td>
</tr>
<tr>
<td>Memory Interface</td>
<td>384-bit GDDR5</td>
<td>384-bit GDDR5</td>
<td>4096-bit HBM2</td>
<td>4096-bit HBM2</td>
</tr>
<tr>
<td>Maximum TDP</td>
<td>244W</td>
<td>250W</td>
<td>250W</td>
<td>300W</td>
</tr>
</tbody>
</table>
Single-Instruction, Multiple-Thread

- SIMT is an execution model where single instruction, multiple data (SIMD) is combined with multithreading.
- The SM creates, manages, schedules, and executes threads in groups of 32 parallel threads called warps.
- A warp start together at the same program address, but they have their own instruction address counter and register state and are therefore free to branch and execute independently.
CUDA

• The NVIDIA GPU architecture is built around a scalable array of multithreaded Streaming Multiprocessors (SMs).
• CUDA (Compute Unified Device Architecture) provides a way how a CUDA program can be executed on any number of SMs.
• A multithreaded program is partitioned into blocks of threads that execute independently from each other.
• A GPU with more multiprocessors will automatically execute the program in less time than a GPU with fewer multiprocessors.
CUDA

A multithreaded CUDA program is shown with eight blocks: Block 0 to Block 7. These blocks are distributed across two GPUs: one with 2 SMs (SM 0 and SM 1) and another with 4 SMs (SM 0, SM 1, SM 2, SM 3). Each SM is further divided into blocks: Block 0 to Block 7.
Grid/Block/Thread

- Threads can be identified using a 1-D, 2-D, or 3-D thread index, forming a 1-D, 2-D, or 3-D block of threads, called a thread block.
- Blocks are organized into a 1-D, 2-D, or 3-D grid of thread blocks.

2-D grid with 2-D thread blocks
Kernel

- CUDA C extends C by allowing the programmer to define C functions, called kernels.

```c
// Kernel definition
__global__ void VecAdd(float* A, float* B, float* C)
{
    int i = threadIdx.x;
    C[i] = A[i] + B[i];
}

int main()
{
    // Kernel invocation with N threads inside 1 thread block
    VecAdd<<<1, N>>>(A, B, C);
}
```

- `threadIdx` is a 3-component vector, so that threads can be identified using a 1-D, 2-D, or 3-D thread index.
Memory Hierarchy

- Each thread has private set of **registers** and **local memory**.
- Each thread block has **shared memory** visible to all threads of the block.
- All threads have access to the same **global memory**.
- There are also two additional read-only memory spaces accessible by all threads (**constant** and **texture memory**).
GPU Programming - Example

• Element by element vector addition

Element by element vector addition

/* Host main routine */
int main(void)
{
    int numElements = 50000;
    size_t size = numElements * sizeof(float);

    // Allocate the host input vectors A and B and output vector C
    float *h_A = (float *)malloc(size);
    float *h_B = (float *)malloc(size);
    float *h_C = (float *)malloc(size);

    // Initialize the host input vectors
    for (int i = 0; i < numElements; ++i)
    {
        h_A[i] = rand()/(float)RAND_MAX;
        h_B[i] = rand()/(float)RAND_MAX;
    }
Element by element vector addition

// Allocate the device input vectors A and B and output vector C
float *d_A = NULL;
cudaMalloc((void **)d_A, size);
float *d_B = NULL;
cudaMalloc((void **)d_B, size);
float *d_C = NULL;
cudaMalloc((void **)d_C, size);

// Copy the host input vectors A and B in host memory to the device
// input vectors in device memory
cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);
Element by element vector addition

// Launch the Vector Add CUDA Kernel
int threadsPerBlock = 256;
int blocksPerGrid = (numElements + threadsPerBlock - 1) / threadsPerBlock;

vectorAdd<<<blocksPerGrid, threadsPerBlock>>>(d_A, d_B, d_C, numElements);

// Copy the device result vector in device memory to the host result vector
// in host memory.
cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);
Element by element vector addition

// Free device global memory
err = cudaFree(d_A);
err = cudaFree(d_B);
err = cudaFree(d_C);

// Free host memory
free(h_A);
free(h_B);
free(h_C);

return 0;
}
/**
* CUDA Kernel Device code
*
* Computes the vector addition of A and B into C. The 3 vectors have the same
* number of elements numElements.
*/
__global__ void vectorAdd(float *A, float *B, float *C, int numElements)
{
    int i = blockDim.x * blockIdx.x + threadIdx.x;

    if (i < numElements)
    {
        C[i] = A[i] + B[i];
    }
}
Intel Xeon Phi
Intel Xeon Phi - Roadmap
**Intel Xeon Phi**

- Intel Xeon Phi coprocessors are designed to **extend the reach** of applications that have demonstrated the ability to fully utilize the scaling capabilities of **Intel Xeon processor-based systems**.
- Code compiled for Xeon processors can be run on an Xeon Phi (Knights Landing).
- For successful parallelization it requires a program with **lots of threads** and operations with **vectors**.
Knights Landing Architecture

36 Tiles connected by 2D Mesh Interconnect
Knights Landing Architecture

• The chip is constituted by **36 tiles** interconnected by 2D mesh.
• The tile has **two Cores** (Atom Silvermont architecture), **two vector processing units** (VPU) and **1M L2 cache**.
• A tile can execute concurrently **4 threads**.
• The tiles are interconnected a **cache-coherent 2D mesh**; which provides a higher bandwidth and lower latency compare to the 1D ring interconnect on Knights corner.
• The mesh enforces **XY routing** rule.
Knights Landing Architecture

• Xeon Phi has 2 types of memory: (i) **MCDRAM** (Multi-channel DRAM) and (ii) **DDR**.

• **MCDRAM** is a *high-bandwidth memory* integrated on the package. There are 8 of them 2 GB each.

• **MCDRAM** can be configured at boot time into one of **three modes**:
  – Cache mode – MCDRAM is a cache for DDR,
  – Flat mode – MCDRAM is a standard memory in the same address space as DDR,
  – Hybrid – a combination

• **DDR** is a *high-capacity memory* which is external to the Knight Landing package.
Vectorization

- Each tile has **two VPUs** (Vector Processing Unit).
- It is the heart of computation. It processes all floating point computations using SSE, AVX, AVX2, …, AVX-512.
- Thus each tile can **execute two 512-bit vector multiple-add instructions per cycle**, i.e. compute 32 double precision resp. 64 single precision floating point operation in each cycle.
# Knights Corner vs. Knights Landing

<table>
<thead>
<tr>
<th>Product Name</th>
<th>Intel® Xeon Phi™ Coprocessor 7120X</th>
<th>Intel® Xeon Phi™ Processor 7290F</th>
</tr>
</thead>
<tbody>
<tr>
<td>Code Name</td>
<td>Knights Corner</td>
<td>Knights Landing</td>
</tr>
<tr>
<td>Lithography</td>
<td>22 nm</td>
<td>14 nm</td>
</tr>
<tr>
<td>Recommended Customer Price</td>
<td>N/A</td>
<td>$6401.00</td>
</tr>
<tr>
<td># of Cores</td>
<td>61</td>
<td>72</td>
</tr>
<tr>
<td>Processor Base Frequency</td>
<td>1.24 GHz</td>
<td>1.50 GHz</td>
</tr>
<tr>
<td>Cache</td>
<td>30.5 MB L2</td>
<td>36 MB L2</td>
</tr>
<tr>
<td>TDP</td>
<td>300 W</td>
<td>260 W</td>
</tr>
<tr>
<td>Max Memory Size (dependent on memory type)</td>
<td>16 GB</td>
<td>384 GB</td>
</tr>
<tr>
<td>Max Memory Bandwidth</td>
<td>352 GB/s</td>
<td>490 GB/s</td>
</tr>
</tbody>
</table>
Offloading

• Choose **highly-parallel sections** of code to run on the coprocessor. Serial code offloaded to the coprocessor will run much slower than on the CPU.

• Intel provides a set of directives to the compiler named “LEO”: **Language Extension for Offload.**

• These directives manage the transfer and execution of portions of code to the device.

```c
#include <omp.h>

#pragma offload target(mic:0)
{
    // a highly-parallel section
}
```
Offloading

- A host and device don't have a shared memory, therefore **variables must be copied** in an explicit or in an implicit way.
- **Implicit copy** is assumed for scalar variables and static arrays.
- **Explicit copy** must be managed by the programmer using clauses defined in the LEO.
- Programmer clauses for explicit copy: `in, out, inout, nocopy`

```c
#pragma offload target(mic) in(data:length(size))
```
void f()
{
    int x = 55;
    int y[10] = {0,1,2,3,4,5,6,7,8,9};
    // x, y sent from CPU
    // To use values computed into y by this offload in next offload,
    // y is brought back to the CPU
    #pragma offload target(mic:0) in(x,y) inout(y)
    { y[5] = 66; }
    // The assignment to x on the CPU
    // is independent of the value of x on the coprocessor
    x = 30;

    // Reuse of x from previous offload is possible using nocopy
    // However, array y needs to be sent again from CPU
    #pragma offload target(mic:0) nocopy(x) in(y)
    { = y[5]; // Has value 66
      = x;     // x has value 55 from first offload
    }
}
Xeon Phi Programming - Demo

• Simple matrix equation \( A = a^* A + B \).

```cpp
int main(int argc, char* argv[]) {

    high_resolution_clock::time_point start = high_resolution_clock::now();
    uint64_t numThreads = 1;

    #pragma offload target(mic: 0)
    {
        init();
        numThreads = omp_get_max_threads();
        #pragma omp parallel
        {
            kernel();
        }
    }

    double totalDuration = duration_cast<duration<double>>(high_resolution_clock::now() - start).count();
    return 0;
}
```
# Simple matrix equation

```c
#define IMCI_ALIGNMENT 64
#define ARR_SIZE 1024*1024
#define ITERS_PER_LOOP 128
#define LOOP_COUNT (64*1024*1024)

#pragma omp declare target
float a;
float fa[ARR_SIZE] __attribute__((aligned(IMCI_ALIGNMENT)));
float fb[ARR_SIZE] __attribute__((aligned(IMCI_ALIGNMENT)));

inline void init()
{
    a = 1.1f;
    #pragma omp simd aligned(fa, fb: IMCI_ALIGNMENT)
    for (uint64_t i = 0; i < ARR_SIZE; ++i)
    {
        fa[i] = ((float) i) + 0.1f;
        fb[i] = ((float) i) + 0.2f;
    }
}
```
inline void kernel()
{
    uint64_t offset = omp_get_thread_num() * ITERS_PER_LOOP;
    for (uint64_t j = 0; j < LOOP_COUNT; ++j)
    {
        #pragma omp simd aligned(fa, fb: IMCI_ALIGNMENT)
        for (uint64_t k = 0; k < ITERS_PER_LOOP; ++k)
            fa[k+offset] = a*fa[k+offset]+fb[k+offset];
    }
} #pragma omp end declare target
References


References
