MIC & GPU Architecture

Dr. Momme Allalen, LRZ, allalen@lrz.de
PRACE PATC: Intel MIC&GPU Programming Workshop
Agenda

- Short Intro to the accelerated computing
- Intel Xeon Phi (MIC)
  - Architecture overview
  - MIC@LRZ
  - Access SuperMIC
- GPU
  - Architecture overview
  - GPU@LRZ
  - Programming GPUs using CUDA C
Why do we need co-processors or “accelerators” on HPC?

• In the past, computers got faster by increasing the clock frequency of the core, but this has now reached its limit mainly due to power requirements and heat dissipation restrictions (unmanageable problem).

• Today, processor cores are not getting any faster, but instead the number of cores per chip increases.

• On HPC, we need a chip which can provide higher computing performance at lower energy.
Why do we need co-processors or “accelerators” on HPC?

• The actual solution is a hybrid system containing both CPUs and “accelerators”, plus other forms of parallelism such as vector instruction support.

• Widely accepted that hybrid systems with accelerators deliver the highest performance, and energy efficient computing in HPC.

• Today!! the accelerated computing is revolutionizing HPC.
Architectures Comparison (CPU vs GPU)

- Large cache and sophisticated flow control minimize latency for arbitrary memory access for serial process.

- Simple flow control and limited cache.
- Devotes more transistors for computing in parallel.
- Same problem executed on many data elements in parallel.
Intel Xeon Phi Coprocessors

- Add-on to CPU based systems (PCIe interface)
- High Power efficiency
- ~ 1 TFLOP/s in DP
- Heterogeneous clustering
- Currently it’s major part of the world’s fastest supercomputer: Tianhe-2 in Guangzhou, China

- Most of the major HPC vendors are supporting Xeon Phi
Architectures Comparison

**CPU**
- Control
- Cache
- DRAM

**MIC**
- ALU
- ALU
- ALU
- DRAM

**GPU**
- DRAM

**General-purpose architecture**

**Power-efficient**
- Multiprocessor X86 design architecture

**Massively data parallel**
Intel MIC Architecture in common with Intel many-core Xeon CPUs

Xeon CPUs:
- X86 architecture
- C, C++ and Fortran
- Standard parallelization libraries
- Similar optimization methods

- PCIe bus connection
- IP-addressable
- Own Linux OS
- 8 GB cached
- Up to 61 x86 64 bit cores
- Up to 1 GHz
- 512 bit SIMD vector registers
- 4 way hyper-threading
- SSE & AVX set for SIMD are not supported
- Intel Initial Many Core Instructions (IMCI).
61 cores interconnected by bidirectional ring interconnect which connects all the cores, L2 caches through a Tag Directory, PCIe client logic, GDDR5 memory controllers … etc.
Network Access

- Network access possible using TCP/IP tools like ssh.
- NFS mounts on Xeon Phi supported.
- Proxy Console /File I/O.

First experiences with the Intel MIC architecture at LRZ, Volker Weinberg and Momme Allalen, inSIDE Vol. 11 No.2 Autumn 2013
### Cache Structure

<table>
<thead>
<tr>
<th>Description</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cache line size</td>
<td>64B</td>
</tr>
<tr>
<td>L1 size</td>
<td>32KB data cache, 32KB instruction code</td>
</tr>
<tr>
<td>L1 latency</td>
<td>1 cycle</td>
</tr>
<tr>
<td>L2 size</td>
<td>512 KB</td>
</tr>
<tr>
<td>L2 ways</td>
<td>8</td>
</tr>
<tr>
<td>L2 latency</td>
<td>15-30 cycles</td>
</tr>
<tr>
<td>Memory → L2 prefetching</td>
<td>hardware and software</td>
</tr>
<tr>
<td>L2 → L1 prefetching</td>
<td>Software only</td>
</tr>
<tr>
<td>Translation Lookside Buffer (TLB)</td>
<td>64 pages of size 4KB (256KB coverage)</td>
</tr>
<tr>
<td>Coverage options (L1, data)</td>
<td>8 pages of size 2MB (16MB coverage)</td>
</tr>
</tbody>
</table>

Intel's Jim Jeffers and James Reinders: *Intel Xeon® Phi™ Coprocessor High Performance Programming*, the first book on how to program this HPC vector chip, is available on Amazon.
Cache Structure

- L2 size depends on how data/code is shared between the cores
  - If no data is shared between cores: **L2 size is 30.5 MB** (61 cores).
  - If every core shares the same data: **L2 size is 512 KB**.
  - Cache coherency across the entire coprocessor.
  - Data remains consistent without software intervention.
## Vector Instruction Sets

<table>
<thead>
<tr>
<th>Instruction Set</th>
<th>Year &amp; Processor</th>
<th>SIMD Width</th>
<th>Data Types</th>
</tr>
</thead>
<tbody>
<tr>
<td>MMX</td>
<td>1997 Pentium</td>
<td>64-bit</td>
<td>8/16/32-bit Int.</td>
</tr>
<tr>
<td>SSE</td>
<td>1999 Pentium III</td>
<td>128-bit</td>
<td>32-bit SP FP</td>
</tr>
<tr>
<td>SSE2</td>
<td>2001 Pentium 4</td>
<td>128-bit</td>
<td>8-64-bit Int., SP&amp;DP FP</td>
</tr>
<tr>
<td>SSE3-SSE4.2</td>
<td>2004-2009</td>
<td>128-bit</td>
<td>Additional instructions</td>
</tr>
<tr>
<td>AVX</td>
<td>2011 Sandy-Bridge</td>
<td>256-bit</td>
<td>SP &amp; DP FP</td>
</tr>
<tr>
<td>AVX2</td>
<td>2013 Haswell</td>
<td>256-bit</td>
<td>Int. &amp; additional instruct</td>
</tr>
<tr>
<td>IMCI</td>
<td>2012 KNC</td>
<td>512-bit</td>
<td>32/64-bit Int., SP &amp; DP FP</td>
</tr>
</tbody>
</table>
SuperMIC cluster configuration:
The cluster consists of 32 nodes dx360 M4
Each node contains
- 2 Ivy-Bridge (2x8cores) host processors E5-2650 of 2.6 GHz
- 2 Intel Xeon Phi coprocessors 5110P
- 2 MLNX CX3 FDR PCIe cards attached to each CPU socket
- Memory per node: 64 GB (host) + 2 *8 GB (Phi)
- SLES11 SP3 with MPSS 3.1.2
- OpenFabrics OFED 1.5.4.1
- Melanox Infiniband FDR14
- 1 xCAT Management node
- All nodes and MICs are accessible through Bridge Interface: i01r13c01, i01r13c01-mic0…
Once you’ve logged into SuperMUC, you can access SuperMIC via:

```
ssh supermic.smu.lrz.de
```
Access SuperMIC
ssh training.srv.mwn.de –l userID
ssh supermuc
ssh supermic.smuc.lrz.de

lu23bik        lu23vet
lu23bim        lu23vob
lu23bip        lu23vof
lu23bud        lu23voh
lu23buf        lu23voj
lu23bug        lu23voq
lu23bul        lu23vov
lu23bun        lu23vox
lu23bur        lu23voz
lu23veq
Interacting with Intel Xeon Phi coprocessors

```bash
user@host$ /sbin/lspci | grep -i "co-processor"
20:00.0 Co-processor: Intel Corporation Device 2250 (rev 11)
8b:00.0 Co-processor: Intel Corporation Device 2250 (rev 11)

user@host$ cat /etc/hosts | grep mic1
10.5.1.128  i01r13a01-mic0.sm.lrz.de  i01r13a01-mic0
10.5.1.129  i01r13a02-mic0.sm.lrz.de  i01r13a02-mic0
...

user@host$ cat /etc/hosts | grep mic1-ib0 | wc -l
32

user@host$ sudo ssh i01r13a01-mic0
ls /
user@host-mic0$ ls /
bin boot dev etc home init lib lib64 lrz media
mnt proc root sbin sys tmp usr var
```
micinfo  a system information query tool

micsmc  a utility for monitoring the physical parameters of Intel Xeon Phi coprocessors: model, memory, core rail temperatures, core frequency, power usage, etc.

Micctrl  a comprehensive configuration tool for the Intel Xeon Phi coprocessor operating system

miccheck  a set of diagnostic tests for the verification of the Intel Xeon Phi coprocessor configuration

micrasd  a host daemon logger of hardware errors reported by Intel Xeon Phi coprocessors

micflash  an Intel Xeon Phi flash memory agent
GPU Hardware Overview and Programming GPUs using CUDA C
What you will learn?

- What is GPUs?
- Write&compile and launch CUDA C kernels.
- Manage GPU memory from host.
- Run parallel kernels in CUDA C.
General Purpose computation using Graphics Processing Units in applications other than 3D graphics

When the science community turned their attention to the GPU predominantly discussed in the computer gaming circles:

- GPU accelerates critical path of application
- Large data arrays and high memory bandwidth
- The same procedure on many elements
- Fine-grain SIMD architecture
- Transistors used to run many floating point operations
Motivation: Why GPU for computing?

Performance Floating-point operation per second
GPUs in HPC
Heterogeneous parallel Computing

- GPU acts as an accelerator to the CPU
  - Most lines of code are executed on the CPU
  - Key computational kernels are executed on the GPU (taking the advantage of the large number of cores and high graphics memory bandwidth)

- Host - The CPU and its memory (host memory)
- Device - The GPU and its memory (device memory)
SIMD & SISD

- SIMD make use of data parallelism.
- We care about SIMD because of area and power efficiency concerns.
Stream Computing

Parallel code (kernel) is launched and executed on a device by many threads.

- Data set decomposed into a stream of elements
- Multiple cores can process multiple elements in parallel
  i.e. many threads running in parallel
## NVidia Tesla GPU

<table>
<thead>
<tr>
<th></th>
<th>Tesla K20X</th>
<th>Tesla K40</th>
<th>Tesla K80</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUDA Cores</td>
<td>2688</td>
<td>2880</td>
<td>2x2496</td>
</tr>
<tr>
<td>SMs</td>
<td>15</td>
<td>15</td>
<td>2x14</td>
</tr>
<tr>
<td>Core Clock (MHz)</td>
<td>732</td>
<td>745</td>
<td>562</td>
</tr>
<tr>
<td>Boost Clock (MHz)</td>
<td>810</td>
<td>875</td>
<td></td>
</tr>
<tr>
<td>Peak SP (TF)</td>
<td>3.93</td>
<td>4.29</td>
<td>8.74</td>
</tr>
<tr>
<td>Peak DP (TF)</td>
<td>1.17</td>
<td>1.43</td>
<td>2.91</td>
</tr>
<tr>
<td>Memory size (GB)</td>
<td>6</td>
<td>12</td>
<td>24</td>
</tr>
<tr>
<td>Memory Clock (GHz)</td>
<td>2.6</td>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>Bandwidth</td>
<td>250 GB/s</td>
<td>288 GB/s</td>
<td>up to 480 GB/s</td>
</tr>
<tr>
<td>Price</td>
<td>~3799 $</td>
<td>~5499 $</td>
<td>~5000 $</td>
</tr>
<tr>
<td>Power</td>
<td>245 W</td>
<td>300 W</td>
<td></td>
</tr>
</tbody>
</table>
Programming GPUs using CUDA

• CUDA = ”Compute Unified Device Architecture“, since Nov. 2006

• Requires an NVIDIA GPU
  • You do not need any GPU experience
  • You do not need any graphics experience
  • You do not need any parallel programming experience
• Driver for loading computation programs into GPU

• You probably need experience with C or C++
• Only “C” plus a few simple extensions
  • compute oriented drivers, language, and tools
Programming GPUs using CUDA

• CUDA is designed for:
  • Heterogeneous architecture
  • Many SIMD parallelism

• CUDA provides:
  • A thread abstraction to deal with SIMD
  • Synchronization
  • Data sharing between small thread groups
Standard C program: Hello, World!

```c
int main ( void ) {
    printf ("Hello, World! \n");
    return 0;
}
```

- runs on the **host**
- NVIDIA’s compiler (nvcc) will not complain about CUDA program with no device code
- at its simplest, CUDA C is just C!
Hello, World! With device Code

```c
__global__ void kernel(void)
{
}

int main ( void ) {
    kernel<<<1,1>>>();
    printf ("Hello, World! \n");
    return 0;
}
```

- Only two addition lines to the standard C program “Hello, World!”
- The triple angle brackets mark indicate that the function will be executed on device

The parameters inside the angle brackets will be discussed later.
__global__ void kernel(void) {
}

• __global__ indicates that a function
  → runs on the device
  → called from the host code

• The compiler nvcc splits the code into host and device components
  → nvcc handles device function like kernel()
  → standard host compiler like gcc handles function like main()
In a Tesla GPU, example: C1060

Streaming processor 240 ALUs (1 mul-add)

- \((1 \text{ mul-add} + 1 \text{ mul}): 240 \times (2 + 1) \times 1.3 \text{ GHz} = 936 \text{ GFLOPS}\)
- 4.0 GB GDDR3, 102 GB/s Mem BW, 4GB/s PCIe BW to CPU
Streaming Multiprocessors (SM)

- Streaming Multiprocessor (SM)
- 8 Streaming Processors (SP)
- 2 Super Function Units (SFU)
- Multi-threaded instruction dispatch
  - Supports up to 2048 threads
- All 8 SPs execute the same instruction
- 3 SMs form building block
- The number of SMs in building block vary from one generation GPUs to another.
- Up to 192 cores on K80
- Up to 2K threads per SM
Access GPU@LRZ
and CUDA-lab1
**LRZ GPU** cluster consists of:
- 4 nodes (T-Blade2-TL) each
  - 2 CPUs Intel 5630
  - 2 GPUs (Tesla M2070)

Login to the cluster
```
ssh lxlogin_gpu
```
Cuda/5.0

**The MAC cluster** consists of:
- 4 nodes each has:
  - 2 GPUs (Tesla M2090)

Login to the cluster
```
ssh mac-login-intel.tum-mac.cos.lrz.de
```
Works with cuda/6.5

**The UV2K cluster:**
- 8 GPUs Tesla K20Xm

Login with: `ssh uv2k`
Works with cuda/6.5
ssh training.srv.mwn.de –l userID
ssh lxlogin1 oder lxlogin2
ssh lxlogin_gpu
salloc --gres=gpu:1 --reservation=gpu_course
NVCC Compiler’s Role: Partition code and compiler for Device

mycode.cu

int main_data;
__shared__ int sdata;

Main () {}       __global__ gfunc() {
  __host__ hfunc() {
    int hdata;
    <<<gfunc(g,b,m)>>>>();
  }
}

__global__ gfunc() {
  int gdata
}

__device__ dfunc() {
  int ddata
}

● compiled by native
● compiler: gcc, icc, cc

compiled by nvcc compiler

Momme Allalen: MIC&GPU@LRZ , April 27-29, 2015
Launch parameters

- Grid dimension (up to 2D)
- Block dimension (up to 3D)
- Optional: stream ID
- Optional: shared memory size
- Kernel `<<grid, block, stream, shared_mem >>>();`

```c
__global__ void filter(int *in, int *out);
...
dim3 grid(16, 16);
dim3 block(16, 16);
filter <<< grid, block >>> (in, out);
```
• Simple example: 1D grid with 4 blocks, each with 64 threads:
  - `gridDim = 4 (gridDim.x, gridDim.y)`
  - `blockDim = 64`
  - `blockIdx` ranges from 0 to 3: `(blockDim.x, blockDim.y, blockDim.z)`
  - `threadIdx` within current block ranges from 0 to 63 (`threadIdx.x, threadIdx.y, threadIdx.z`)

  ```
  int idx = blockIdx.x * blockDim.x + threadIdx.x;
  ```
  defining the thread's position within the grid and used to access memory in vector format

For simple case of 1D grid of blocks & 1D set of threads, e.g. `blockIdx.x=1, threadIdx.x=33`
• If we want to use a 2D set of threads, then
  - blockDim.x, blockDim.y give the dimensions
  - threadIdx.x, threadIdx.y give the threads indices

• And to launch the kernel we would use something like
  - dim3 nthreads (16, 4);
  - kernel <<<nbblocks, nthreads>>> (args);

  where dim3 is a special CUDA data type with 3 components, x,y,z each initialized to 1

• And a similar approach is used for 3D threads.
Thread Batching grids and Blocks

- A kernel is executed as a grid of thread blocks
  - All threads share data memory space
- A thread block is a batch of threads that can cooperate with each other by:
  - Synchronizing their execution
    → For hazard-free shared memory accesses
  - Efficiently sharing data through a low latency shared memory
- Two threads from two different blocks cannot cooperate
- All blocks define the grid
- All blocks execute same program (kernel)
How we do run code in parallel on the device?

The Solution lies in the parameters between the triple angle brackets:

\[
\text{add}^{\text{<<<1, 1>>>}}(\text{args});
\]

\[
\text{add}^{\text{<<<\text{N}, 1>>>}}(\text{args});
\]

Instead of executing \text{add} once, \text{add} executed \text{N} times in parallel.


```c
__global__ void add(int *a, int *b, int *c){
    *c = *a + *b;
}
```

- add () runs on the device…so a, b, and c must point to the device memory

- How do we allocate memory on the GPU?
Memory
Memory Architecture

Host
- Chipset
- DRAM

Device
- DRAM
  - local
- Global
- Constant
- Texture

GPU
- Multiprocessor
- Multiprocessor
- Registers
- Shared memory
- Constant and Texture
- Caches
Memory Architecture

Local Memory: per-thread
private per thread
Auto variables, register spill

Shared memory: per-Block
-Shared by threads of the same block
-Inter-thread communication

Global memory: per-application
-Shared by all threads
-Inter-Grid communication
Global Memory

- Main means of communicating R/W data between host and device
- Contents visible to all threads
- Long latency access
CUDA API – Memory Allocation

- `cudaMalloc()`
  - Allocates object in the device Global Memory
  - Requires two parameters
    - Address of a pointer to the allocated object
    - Size of allocated object

- `cudaFree()`
  - Frees object from device Global Memory
    - Pointer to freed object

Main means of communicating R/W data between host and device
Device Memory Allocation Example

Code example:
Allocate a 1024 * 1024 single precision float matrix

```c
#define MATRIX_SIZE 1024*1024
float* MyMatrixOnDevice;
int size = MATRIX_SIZE * sizeof(float);
cudaMalloc((void**) &MyMatrixOnDevice, size);
cudaFree(MyMatrixOnDevice);
```
cudaMallocHost() allows allocation of page-locked ("pinned") host memory

- Enables highest cudaMemcpy performance
  - 5.2 GB/s on PCI-e x8 Gen2
  - up to 16 GB/s on PCI-e x16 Gen3

See the "bandwidth Test" CUDA SDK sample

Use with caution!!
- Allocating too much page-locked memory can reduce overall system performance
- Test your systems and apps to learn their limits
Transfer and Memory allocation functions

- `cudaMalloc()` allocates linear memory. It can also allocate through `cudaMallocPitch()` and `cudaMalloc3D()` (recommended for allocations of 2D or 3D arrays).
- `cudaFree()` frees using memory.
- `cudaMemcpy()` transfers between host memory and device memory.
- `cudaMemcpy2D()` and `cudaMemcpy3D()` functions: copy between 2D arrays and other regions of device memory.
Memory Spaces

- CPU and GPU have separate memory spaces
  - Data is moved across PCIe bus
  - Use functions to allocate/set/copy memory on GPU
    - very similar to corresponding C functions

- Pointers are just addresses
  - Can’t tell from the pointer value whether the address is on CPU or GPU
  - Dereferencing CPU pointer in code that executes on the GPU will likely crash
  - Dereferencing GPU pointer in code that executes on the CPU will likely crash
Host-Device
Data Transfers

- Host (CPU) manages device (GPU) memory
CUDA Host-Device Data Transfer

- **cudaMemcpy()**
  - memory data transfer
  - Requires 4 parameters
    - Pointer to source
    - Pointer to destination
    - Number of bytes copied
    - Type of transfer
      - Host to Host
      - Host to Device
      - Device to Host
      - Device to Device

- Asynchronous variant supported on 1.1+HW
CUDA Host-Device Data Transfer Examples

Code example:
- Transfer a 1024 * 1024 single precision float matrix
- MyMatrixOnHost is in host memory and MyMatrixOnDevice is in device memory
- cudaMemcpyHostToDevice and cudaMemcpyDeviceToHost are symbolic constants

```c
cudaMemcpy(MyMatrixOnDevice, MyMatrixOnHost, size, cudaMemcpyHostToDevice);
```

```c
cudaMemcpy(MyMatrixOnHost, MyMatrixOnDevice, size, cudaMemcpyDeviceToHost);
```
cudaMemcpy2D(data_d, pitch, data_h
width*sizeof(float), sizeof(float)*width,
height, cudaMemcpyHostToDevice );
CUDA-lab3: Vectors 1D addition
#include <stdio.h>
__global__ void kernel(int* a) {
    int idx = blockIdx.x*blockDim.x+ threadIdx.x;
    a[idx] = 7;
}

int main() {
    int dimx = 16, num_bytes = dimx*sizeof(int);
    int*d_a = 0, *h_a = 0;  // device and host pointers
    h_a = (int*)malloc(num_bytes);
    cudaMalloc((void**)&d_a, num_bytes);
    if (0 == h_a|| 0 == d_a) {
        printf("couldn't allocate memory\n");
        return 1;
    }
    cudaMemcpy(d_a, h_a, num_bytes, cudaMemcpyDeviceToHost);
    kernel<<<grid, block>>>(d_a);
    cudaMemcpy(h_a, d_a, num_bytes, cudaMemcpyDeviceToHost);
    for(int i=0; i< dimx; i++)
        printf("%d\n", h_a[i]);
    free(h_a);
    cudaFree(d_a);
    return 0;
}
Kernel Code (Executed on the GPU)

```c
__global__ void kernel(int* a)
{
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    a[idx] = 7;
}
```
Kernel Variations and output

```c
__global__ void kernel(int* a)
{
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    a[idx] = 7;                  // Output: 7777777777777777
}

__global__ void kernel(int* a)
{
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    a[idx] = blockIdx.x;         // Output: 0000111122223333
}

__global__ void kernel(int* a)
{
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    a[idx] = threadIdx.x;        // Output: 0123012301230123
}
```
A simple matrix multiplication example that illustrates the basic features of memory and thread management in CUDA programs.

- Leave shared memory usage until later
- Local, register usage
- Thread ID usage
- Memory data transfer API between host and device
- Assume square matrix for simplicity
Programming Model: Square Matrix Multiplication Example

- **P = M \times N** of size WIDTH x WIDTH
- **Without tiling:**
  - One thread calculates one element of **P**
  - **M** and **N** are loaded WIDTH times from global memory
Step 1: A simple Host version in C

// Matrix multiplication on the (CPU) host!
void MatrixMulOnHost(float* M, float* N, float* P, int Width) {
    for (int i = 0; i < Width; ++i)
        for (int j = 0; j < Width; ++j) {
            float sum = 0;
            for (int k = 0; k < Width; ++k) {
                float a = M[i * width + k];
                float b = N[k * width + j];
                sum += a * b;
            }
            P[i * Width + j] = sum;
        }
}
Using CUDA

- Allocate CPU memory for n integers
- Allocate GPU memory for n integers
- Initialize GPU memory to 0s
- Copy from GPU to CPU
- Print the values
Step 2: Input Matrix Data Transfer (Host-side Code)

```c
void MatrixMulOnDevice(float* M, float* N, float* P, int Width)
{
    int size = Width * Width * sizeof(float);
    float* Md, Nd, Pd;
    ...
    1. // Allocate and Load M, N to device memory
        cudaMalloc(&Md, size);
        cudaMemcpy(Md, M, size, cudaMemcpyHostToDevice);
        cudaMalloc(&Nd, size);
        cudaMemcpy(Nd, N, size, cudaMemcpyHostToDevice);
    // Allocate P on the device
        cudaMalloc(&Pd, size);
```
Step3: Output Matrix Data Transfer (Host-side Code)

2. // Kernel invocation code – to be shown later
   ...
3. // Read P from the device
   cudaMemcpy(P, Pd, size, cudaMemcpyDeviceToHost);

   // Free device matrices
   cudaFree(Md);
   cudaFree(Nd);
   cudaFree (Pd);
}
Step4: Kernel Function

// Matrix multiplication kernel – per thread code

__global__ void MatrixMulKernel(float* Md, float* Nd, float* Pd, int Width)
{
    // Pvalue is used to store the element of the matrix
    // that is computed by the thread
    float Pvalue = 0;
Step 4: Kernel Function

for (int k = 0; k < Width; ++k) {
    float Melement = Md[threadIdx.y*Width+k];
    float Nelement = Nd[k*Width+threadIdx.x];
    Pvalue += Melement * Nelement;
}

Pd[threadIdx.y*Width+threadIdx.x] = Pvalue;
Step 5: Kernel Invocation
(Host-side Code)

// Setup the execution configuration
    dim3 dimGrid(1, 1);
    dim3 dimBlock(Width, Width);

// Launch the device computation threads

    MatrixMulKernel<<<dimGrid, dimBlock>>>(Md, Nd, Pd, Width);
CUDA-lab4: Vectors 2D addition
Things you need to remember
CUDA Programming Model

• The GPU is viewed as a compute device that
  • is a coprocessor to the CPU or host similarly to Xeon Phi
  • has its own DRAM (device memory)
  • runs many threads in parallel
• Data-parallel portions of an application are executed on the device as kernels which run in parallel on many threads
• Differences between GPU and CPU threads
  • GPU needs 1000s of threads for full efficiency
  • Up to 1024 threads per block are alive simultaneously
  • Multi-core CPU needs only a few.
CUDA Programming Model

- The master process which runs on the CPU performs the following steps:
  - Initialises the device.
  - Allocates memory in host and device.
  - Copies data from host to device memory.
  - Launches multiple instances of execution “kernel” on device.
  - Copies data from device memory to host.
  - Repeats 3-5 as needed.
  - De-allocates all memory and terminates.
Launching Kernels

- Kernel launch parameters:
  - Grid dimensions (up to 2D), dim3type
  - Thread block dimensions (up to 3D), dim3type
  - Other optional parameters (0 by default):
    - Shared memory allocation (number of bytes per block) for \_\_shared\_\_array declared without size
    - Stream ID

```c
dim3 grid(16, 16);
dim3 block(16, 16);
kernel<<<grid, block, 0, 0>>>(...);
kernel<<<32, 512>>>(...);
```
Kernel with 2D indexing

```c
__global__ void kernel(int* a, intdimx, intdimy)
{
    int ix = blockIdx.x* blockDim.x + threadIdx.x;
    int iy = blockIdx.y* blockDim.y + threadIdx.y;
    int idx= iy* dimx + ix;
    a[idx] = a[idx] + 1;
}
```
CUDA 6 Production Release is now available!

Includes features:

- **Unified Memory**: simplifies programming by enabling applications to access the CPU and GPU memory without the need to manually copy data.
- **Drop-in Libraries**: automatically accelerate the linear algebra and FFTs in your applications by replacing CPU-only libraries with GPU-accelerated libraries.
- **Multi-GPU Scaling**: the re-designed BLAS GPU library automatically scales performance across up to 8 GPUs in a single node.
Unified Memory in CUDA 6

CPU and GPU memories are physically distinct and separated by the PCI-Express bus.

Wrap up

- Accelerated computing is revolutionizing HPC industries.

- Once code is ported to accelerator based system, usually much more optimization work is required to achieve high sustained performance.

- Xeon Phi based systems: codes using standard parallelization tools are quickly ported, only for a good performance consider optimizing your code by SIMD vectorisation.

- On CUDA and NVIDIA GPUs the traditional language (C, C++, Fortran.. etc) is not enough, need extensions, directives, or new language.
References

- First experiences with the Intel MIC architecture at LRZ, Volker Weinberg and Momme Allalen, inSIDE Vol. 11 No.2 Autumn 2013
- V. Weinberg (Editor) et al., Best Practice Guide - Intel Xeon Phi, [http://www.prace-project.eu/Best-Practice-Guide-Intel-Xeon-Phi-HTML](http://www.prace-project.eu/Best-Practice-Guide-Intel-Xeon-Phi-HTML) and references therein