GPGPU Architectures and CUDA C

Swar endu Biswas

An Introductory Course on High-Performance Computing in Engineering
Indian Institute of Technology Kanpur
30th Sep Oct 2019

Content influenced by many excellent references, see References slide for acknowledgements.
A Brief History of GPUs

• Many real-world applications are compute-intensive and data-parallel
  • They need to process a lot of data, mostly floating-point operations
  • For example, real-time high-definition graphics applications such as your favorite video games
  • Iterative kernels which update elements according to some fixed pattern called a stencil
A Brief History of GPUs

• This spurred the need for a highly-parallel computational device with high computational power and memory bandwidth
  • CPUs are more complex devices catering to a wider audience

• GPUs are now used in different applications
  • Game effects, computational science simulations, image processing and machine learning, linear algebra

• Several GPU vendors like NVIDIA, AMD, Intel, QualComm, ARM, etc.
Key Insights in the GPGPU Architecture

• GPUs are suited for compute-intensive data-parallel applications
  • The same program is executed for each data element
  • High arithmetic intensity which can hide latency of memory accesses
  • Less complex control flow

• Much more transistors or real-estate is devoted to computation rather than data caching and control flow
Floating-Point Operations per Second for the CPU and GPU

Theoretical Peak Performance, Single Precision

GFLOP/sec

End of Year


INTEL Xeon CPUs
NVIDIA GeForce GPUs
AMD Radeon GPUs
INTEL Xeon Phis

Xeon Phi 7120 (KNL)
Xeon Phi 7250 (KNL)
NVIDIA Titan X
Tesla K40
FirePro W9100
FirePro S9150
HD 9970
HD 970
HD 8970
HD 8970
HD 7970
HD 7970
HD 6970
HD 6970
HD 5970
HD 5970
HD 4970
HD 4970
8900 GTS
8900 GTS
GTX 590
GTX 590
GTX 580
GTX 580
GTX 480
GTX 480
GTX Titan
GTX Titan
GTX Titan
Memory Bandwidth for CPU and GPU
# High-end CPU-GPU Comparison

<table>
<thead>
<tr>
<th></th>
<th>Xeon 8180M</th>
<th>Titan V</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Cores</strong></td>
<td>28</td>
<td>5120 (+ 640)</td>
</tr>
<tr>
<td><strong>Active threads</strong></td>
<td>2 per core</td>
<td>32 per core</td>
</tr>
<tr>
<td><strong>Frequency</strong></td>
<td>2.5 (3.8) GHz</td>
<td>1.2 (1.45) GHz</td>
</tr>
<tr>
<td><strong>Peak performance (SP)</strong></td>
<td>4.1? TFlop/s</td>
<td>13.8 TFlop/s</td>
</tr>
<tr>
<td><strong>Peak mem. bandwidth</strong></td>
<td>119 GB/s</td>
<td>653 GB/s</td>
</tr>
<tr>
<td><strong>Maximum power</strong></td>
<td>205 W</td>
<td>250 W*</td>
</tr>
<tr>
<td><strong>Launch price</strong></td>
<td>$13,000</td>
<td>$3000*</td>
</tr>
</tbody>
</table>

**Release dates**
- Xeon: Q3’17
- Titan V: Q4’17
GPGPU

• Multi-core chip
• SIMD execution within a single core (many ALUs performing the same instruction)
• Multi-threaded execution on a single core (multiple threads executed concurrently by a core)
Advantages of a GPU

• Performance
  • 3.4x as many operations executed per second

• Main memory bandwidth
  • 5.5x as many bytes transferred per second

• Cost- and energy-efficiency
  • 15x as much performance per dollar
  • 2.8x as much performance per watt

(based on peak values)

• GPU’s higher performance and energy efficiency are due to different allocation of chip area
  • High degree of SIMD parallelism, simple in-order cores, less control/sync. logic, lower cache/scratchpad capacity
GPU Disadvantages

• Clearly, we should be using GPUs all the time
  • So why aren’t we?

• GPUs can only execute some types of code fast
  • Need lots of data parallelism, data reuse, & regularity
  • SIMD parallelism is not well suited for all algorithms

• GPUs are harder to program and tune than CPUs
  • Mostly because of their architecture
  • Fewer tools and libraries exist
GPU Architecture

• GPUs consist of **Streaming Multiprocessors (SMs)**
  • Up to 80 SMs per chip (run blocks)
• SMs contain **Processing Elements (PEs)**
  • Up to 64 PEs per SM (run threads)
Scalability of GPU Architecture

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-Enabled NVIDIA GPUs

<table>
<thead>
<tr>
<th></th>
<th>Embedded</th>
<th>Consumer desktop/laptop</th>
<th>Professional Workstation</th>
<th>Data Center</th>
</tr>
</thead>
<tbody>
<tr>
<td>Turing (Compute capabilities 7.x)</td>
<td>DRIVE/JETSON AGX Xavier</td>
<td>GeForce 2000 Series</td>
<td>Quadro RTX Series</td>
<td>Tesla T Series</td>
</tr>
<tr>
<td>Volta (Compute capabilities 7.x)</td>
<td>DRIVE/JETSON AGX Xavier</td>
<td></td>
<td></td>
<td>Tesla V Series</td>
</tr>
<tr>
<td>Pascal (Compute capabilities 6.x)</td>
<td>Tegra X2</td>
<td>GeForce 1000 Series</td>
<td>Quadro P Series</td>
<td>Tesla P Series</td>
</tr>
<tr>
<td>Maxwell (Compute capabilities 5.x)</td>
<td>Tegra X1</td>
<td>GeForce 900 Series</td>
<td>Quadro M Series</td>
<td>Tesla M Series</td>
</tr>
<tr>
<td>Kepler (Compute capabilities 3.x)</td>
<td>Tegra K1</td>
<td>GeForce 600/700 Series</td>
<td>Quadro K Series</td>
<td>Tesla K Series</td>
</tr>
</tbody>
</table>
Block Scalability

- Hardware can assign blocks to SMs in any order
  - A kernel with enough blocks scales across GPUs
  - Not all blocks may be resident at the same time
What is CUDA?

• It is general purpose **parallel computing platform** and programming model that leverages the parallel compute engine in NVIDIA GPUs
  • Introduced in 2007 with NVIDIA Tesla architecture
  • CUDA C, C++, Fortran, PyCUDA are language systems built on top of CUDA

• **Three key abstractions** in CUDA
  • Hierarchy of thread groups
  • Shared memories
  • Barrier synchronization
CUDA Programming Model

- Helps fine-grained data parallelism and thread parallelism, nested within coarse-grained data parallelism and task parallelism

1. Partition the problem into coarse sub-problems that can be solved independently
2. Assign each sub-problem to a “block” of threads to be solved in parallel
3. Each sub-problem is also decomposed into finer work items that are solved in parallel by all threads within the “block”
Heterogeneous Computing

**Host**
- CPU and its memory (host memory)

**Device**
- GPU and its memory (device memory)
Heterogeneous Computing

```
#include <iostream>
#include <algorithm>
using namespace std;

#define N        1024
#define RADIUS   3
#define BLOCK_SIZE 16

__global__ void stencil_1d(int *in, int *out) {
    __shared__ int temp[BLOCK_SIZE + 2 * RADIUS];
    int gindex = threadIdx.x + blockIdx.x * blockDim.x;
    int lindex = threadIdx.x + RADIUS;

    // Read input elements into shared memory
    temp[lindex] = in[gindex];
    if (threadIdx.x < RADIUS) {
        temp[lindex - RADIUS] = in[gindex - RADIUS];
        temp[lindex + BLOCK_SIZE] = in[gindex + BLOCK_SIZE];
    }

    __syncthreads();

    // Apply the stencil
    int result = 0;
    for (int offset = -RADIUS; offset <= RADIUS; offset++)
        result += temp[lindex + offset];

    // Store the result
    out[gindex] = result;
}

void fill_ints(int *x, int n) {
    fill_n(x, n, 1);
}

int main(void) {
    int *in, *out;
    // host copies of a, b, c
    int *d_in, *d_out;
    // device copies of a, b, c
    int size = (N + 2*RADIUS) * sizeof(int);

    // Alloc space for host copies and setup values
    in = (int*)malloc(size);
    fill_ints(in, N + 2*RADIUS);
    out = (int*)malloc(size);
    fill_ints(out, N + 2*RADIUS);

    // Alloc space for device copies
    cudaMalloc((void**)&d_in, size);
    cudaMalloc((void**)&d_out, size);

    // Copy to device
    cudaMemcpy(d_in, in, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_out, out, size, cudaMemcpyHostToDevice);

    // Launch stencil_1d() kernel on GPU
    stencil_1d<<<N/BLOCK_SIZE,BLOCK_SIZE>>>(d_in + RADIUS, d_out + RADIUS);

    // Copy result back to host
    cudaMemcpy(out, d_out, size, cudaMemcpyDeviceToHost);

    // Clean up
    free(in); free(out);
    cudaFree(d_in);
    cudaFree(d_out);

    return 0;
}
```

serial code

parallel fn

parallel code

serial code
Heterogeneous Computing

```cpp
#include <iostream>
#include <algorithm>
using namespace std;

#define N          1024
#define RADIUS     3
#define BLOCK_SIZE 16

__global__
void stencil_1d(
  int *in,
  int *out) {

  __shared__ int temp[BLOCK_SIZE + 2 * RADIUS];

  int gindex = threadIdx.x + blockIdx.x * blockDim.x;
  int lindex = threadIdx.x + RADIUS;

  // Read input elements into shared memory
  temp[lindex] = in[gindex];
  if (threadIdx.x < RADIUS) {
    temp[lindex - RADIUS] = in[gindex - RADIUS];
    temp[lindex + BLOCK_SIZE] = in[gindex + BLOCK_SIZE];
  }

  // Synchronize (ensure all the data is available)
  __syncthreads();

  // Apply the stencil
  int result = 0;
  for (int offset = -RADIUS; offset <= RADIUS; offset++)
    result += temp[lindex + offset];

  // Store the result
  out[gindex] = result;
}

void fill_ints(
  int *x,
  int n) {
  fill_n(x, n, 1);
}

int main(void) {
  int *in, *out;

  // host copies of a, b, c
  int *d_in, *d_out;

  // device copies of a, b, c
  int size = (N + 2*RADIUS) * sizeof(int);

  // Alloc space for host copies and setup values
  in  = (int*)malloc(size);
  fill_ints(in,  N + 2*RADIUS);
  out = (int*)malloc(size);
  fill_ints(out, N + 2*RADIUS);

  // Alloc space for device copies
  cudaMalloc((void**)&d_in,  size);
  cudaMalloc((void**)&d_out, size);

  // Copy to device
  cudaMemcpy(d_in,  in,  size, cudaMemcpyHostToDevice);
  cudaMemcpy(d_out, out,  size, cudaMemcpyHostToDevice);

  // Launch stencil_1d() kernel on GPU
  stencil_1d<<<N/BLOCK_SIZE,BLOCK_SIZE>>>(d_in + RADIUS, d_out + RADIUS);

  // Copy result back to host
  cudaMemcpy(out, d_out, size, cudaMemcpyDeviceToHost);

  // Cleanup
  free(in); free(out);
  cudaFree(d_in);
  cudaFree(d_out);

  return 0;
}
```

- **Serial Code**: The serial code is written in C++ and includes the main function and the `fill_ints` function, which is used to fill an array with a specific value.
- **Parallel Code**: The parallel code is written in CUDA and utilizes NVIDIA GPUs. It includes the `stencil_1d` function, which is applied to an array using a stencil, and the `main` function, which manages the data transfer between the host and device.
Simple Processing Flow

CPU
Bridge
CPU Memory

PCI Bus

GigaThread™
Interconnect
L2
DRAM
Simple Processing Flow

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
Simple 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 Extensions for C/C++

• Function launch
  • Calling functions on GPU
• Memory management
  • GPU memory allocation, copying data to/from GPU
• Declaration qualifiers
  • __device, __shared, __local, __global
• Special instructions
  • Barriers, fences, etc.
• Keywords
  • threadIdx, blockIdx, blockDim
Hello World with CUDA

```c
#include <stdio.h>
#include <cuda.h>

__global__ void hwkernel()
{
    printf("Hello world!\n");
}

int main()
{
    hwkernel<<<1, 1>>>()
}
```

$ nvcc hell-world.cu

$ ./a.out
#include <stdio.h>
#include <cuda.h>

__global__ void hwkernel() {
    printf("Hello world!\n");
}

int main() {
    hwkernel<<<1, 1>>>();
    cudaDeviceSynchronize();
}

$ nvcc hell-world.cu
$ ./a.out
Hello world!
$
Hello World with CUDA

```c
#include <stdio.h>
#include <cuda.h>

__global__ void hwkernel() {
    printf("Hello world!\n");
}

int main() {
    hwkernel<<<1, 32>>>();
    cudaThreadSynchronize();
}
```

$ nvcc hell-world.cu

$ ./a.out

Hello world!
Hello world!
Hello world!
Hello world!
...
...
$
Kernels

• Special functions that a CPU can call to execute on the GPU
  • Executed N times in parallel by N different CUDA threads
• CPU can continue processing while GPU runs kernel

• Each thread will execute VecAdd()}
• Each thread has a unique thread ID that is accessible within the kernel through the built-in threadIdx variable

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

int main() {
    ...
    // Kernel invocation with N threads
    VecAdd<<<1, N>>>(A, B, C);
}
Kernels

• GPU spawns m blocks with n threads (i.e., m*n threads total) that run a copy of the same function

• Kernel call returns when all threads have terminated
Function Declarations in CUDA

<table>
<thead>
<tr>
<th>Function Declaration</th>
<th>Executed on</th>
<th>Callable from</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>device</strong> float deviceFunc()</td>
<td>Device</td>
<td>Device</td>
</tr>
<tr>
<td><strong>global</strong> void kernelFunc()</td>
<td>Device</td>
<td>Host</td>
</tr>
<tr>
<td><strong>host</strong> float hostFunc()</td>
<td>Host</td>
<td>Host</td>
</tr>
</tbody>
</table>

__global__ define a kernel function, must return void
Variable Type Qualifiers in CUDA

<table>
<thead>
<tr>
<th>Variable Type Qualifiers</th>
<th>Memory</th>
<th>Scope</th>
<th>Lifetime</th>
</tr>
</thead>
<tbody>
<tr>
<td>Int localVar;</td>
<td>Register</td>
<td>Thread</td>
<td>Thread</td>
</tr>
<tr>
<td><strong>device</strong> <strong>local</strong> int localVar;</td>
<td>Local</td>
<td>Thread</td>
<td>Thread</td>
</tr>
<tr>
<td><strong>device</strong> <strong>shared</strong> int sharedVar;</td>
<td>Shared</td>
<td>Block</td>
<td>Block</td>
</tr>
<tr>
<td><strong>device</strong> int globalVar;</td>
<td>Global</td>
<td>grid</td>
<td>Application</td>
</tr>
</tbody>
</table>

- Automatic variables without any qualifier reside in a register
  - Except arrays that reside in local memory
- Pointers can only point to memory allocated or declared in global memory
Typical CUDA Program Flow

1. Load data into CPU memory
   - fread/rand

2. Copy data from CPU to GPU memory
   - cudaMemcpy(..., cudaMemcpyHostToDevice)

3. Call GPU kernel
   - mykernel<<<x, y>>>(...)

4. Copy results from GPU to CPU memory.
   - cudaMemcpy(..., cudaMemcpyDeviceToHost)

5. Use results on CPU
Thread Hierarchy

• **threadIdx** is a 3-component vector
  - Thread index can be 1D, 2D, or 3D
  - Thread blocks as a result can be 1D, 2D, or 3D

• How to find out the relation between thread ids and threadIdx

• 1D: \( \text{tid} = \text{threadIdx}.x \)

• 2D block of size \((D_x, D_y)\): thread ID of a thread of index \((x, y)\) is \((x + y D_x)\)

• 3D block of size \((D_x, D_y, D_z)\): thread ID of a thread of index \((x, y, z)\) is \((x + y D_x + z D_x D_y)\)
Thread Hierarchy

• Threads in a block reside on the same core, max 1024 threads in a block
• Thread blocks are organized into 1D, 2D, or 3D grids
  • Grid dimension is given by gridDim variable
• Identify block within a grid with the blockIdx variable
  • Block dimension is given by blockDim variable
// Kernel definition
__global__ void MatAdd(float A[N][N], float B[N][N], float C[N][N]) {
  int i = threadIdx.x;
  int j = threadIdx.y;
  C[i][j] = A[i][j] + B[i][j];
}

int main() {
  ...
  // Kernel invocation with one block of N * N * 1 threads
  int numBlocks = 1;
  dim3 threadsPerBlock(N, N);
  MatAdd<<<numBlocks, threadsPerBlock>>>(A, B, C);
  ...
}
Code Snippet

// Kernel definition
__global__ void MatAdd(float A[N][N], float B[N][N], float C[N][N]) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    int j = blockIdx.y * blockDim.y + threadIdx.y;
    if (i < N && j < N)
        C[i][j] = A[i][j] + B[i][j];
}

int main() {
    ...
    // Kernel invocation
    dim3 threadsPerBlock(16, 16);
    dim3 numBlocks(N / threadsPerBlock.x, N / threadsPerBlock.y);
    MatAdd<<<numBlocks, threadsPerBlock>>>(A, B, C);
    ...
}
Thread Warps

• A set of consecutive threads (currently 32) that execute in SIMD fashion

• Warps are scheduling units in an SM
  • Part of the NVIDIA implementation, not the programming model

• Warps share an instruction stream

• Individual threads in a warp have their own instruction address counter and register state

• Prior to Volta, warps used a single shared program counter
  • Branch divergence occurs only within a warp
Thread Warps

• Warp threads are fully synchronized
  • There is an implicit barrier after each step/instruction

• If 3 blocks are assigned to an SM and each block has 256 threads, how many warps are there in an SM?
  • Each Block is divided into 256/32 = 8 Warps
  • There are 8 * 3 = 24 warps
SIMT Architecture

• Single instruction multiple threads
• Very similar in flavor to SIMD
  • In SIMD, you need to know the vector width and possibly use
• You can say that SIMT is SIMD with multithreading
  • You rarely need to know the number of cores in a GPU
Mapping Blocks and Threads

• When a CUDA program on the host CPU invokes a kernel grid, the blocks of the grid are enumerated and distributed to multiprocessors with available execution capacity

• The threads of a thread block execute concurrently on one multiprocessor, and multiple thread blocks can execute concurrently on one multiprocessor

• As thread blocks terminate, new blocks are launched on the vacated multiprocessors
Thread Life Cycle in a GPU

• A grid is launched on the GPU
• Thread blocks are serially distributed to all the SMs
  • Potentially >1 Thread Block per SM
• Each SM launches Warps of threads
  • 2 levels of parallelism
• SM schedules and executes Warps that are ready to run
  • Warps whose next instruction has its operands ready for consumption are eligible for execution
  • Zero-overhead scheduling
• As Warps and thread blocks complete, resources are freed
  • SPA can distribute more thread blocks
Question

mykernel<<<numBlks, thrdsBlk>>>()

dim3 thrdsBlk(x, y, z);
dim3 numBlks(p, q);
const int Nx = 11; // not a multiple of threadsPerBlock.x
const int Ny = 5; // not a multiple of threadsPerBlock.y

DIRECTORY

dim3 threadsPerBlock(4, 3, 1);
dim3 numBlocks(x, y, z);

// assume A, B, C are allocated Nx x Ny float arrays
matrixAdd<<<numBlocks, threadsPerBlock>>>(A, B, C);
const int Nx = 11; // not a multiple of threadsPerBlock.x
const int Ny = 5; // not a multiple of threadsPerBlock.y

dim3 threadsPerBlock(4, 3, 1);
dim3 numBlocks((Nx+threadsPerBlock.x-1)/threadsPerBlock.x, 
(Ny+threadsPerBlock.y-1)/threadsPerBlock.y, 
1);

// assume A, B, C are allocated Nx x Ny float arrays
// this call will cause execution of 72 threads
// 6 blocks of 12 threads each
matrixAdd<<<numBlocks, threadsPerBlock>>>(A, B, C);
Question

```c
__global__ void matrixAdd(float A[Ny][Nx],
float B[Ny][Nx], float C[Ny][Nx]) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    int j = blockIdx.y * blockDim.y + threadIdx.y;
    // guard against out of bounds array access
    if (i < Nx && j < Ny)
        C[j][i] = A[j][i] + B[j][i];
}
```
Memory Organization

• Host and device maintain their own separate memory spaces
• A variable in CPU memory cannot be accessed directly in a GPU kernel
• A programmer needs to maintain copies of variables
• It is programmer's responsibility to keep them in sync
Memory Hierarchy
Vector Addition

// Device code
__global__ void VecAdd(float* A, float* B, float* C, int N) {
    int i = blockDim.x * blockIdx.x + threadIdx.x;
    if (i < N)
        C[i] = A[i] + B[i];
}

// Host code
int main() {
    const int N = (1<<24);
    size_t size = N * sizeof(float);
    float* h_A = (float*)malloc(size);
    float* h_B = (float*)malloc(size);
    float* h_C = (float*)malloc(size);

    // Allocate vectors in host memory
    for (int i = 0; i < N; i++) {
        h_A[i] = i + 10;
        h_B[i] = (N - i) * 0.5;
        h_C[i] = 0.0;
    }
}
Vector Addition

// Allocate vectors in device memory
float* d_A;
cudaMalloc(&d_A, size);
float* d_B;
cudaMalloc(&d_B, size);
float* d_C;
cudaMalloc(&d_C, size);

// Copy from host to device memory
cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);

// Invoke kernel
int threadsPerBlock = 256;
int blocksPerGrid = (N + threadsPerBlock - 1) / threadsPerBlock;
VecAdd<<<blocksPerGrid, threadsPerBlock>>>(d_A, d_B, d_C, N);

// Copy result from device to host memory
cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);

// Free memory
cudaFree(d_A); cudaFree(d_B); cudaFree(d_C);
free(h_A); free(h_B); free(h_C);
Device Memory Management

- Device memory can be allocated either as linear memory or CUDA arrays
  - We will focus on linear memory
- Allocated by `cudaMalloc()` and freed by `cudaFree()`
- Data transfer between host and device is by `cudaMemcpy()`
- Shared memory is allocated using the `__shared__` memory space specifier
  - Supposed to be faster than global memory
Memory Hierarchy

• There are also two additional read-only memory spaces accessible by all threads: the constant and texture memory spaces
  • These are carried over from graphics shading languages
• The global, constant, and texture memory spaces are optimized for different memory usages
• Global, constant, and texture memory spaces are persistent across kernel launches by the same application
Synchronization Constructs in CUDA

• `__syncthreads()`
  • Barrier: wait for all threads in the block to his this point

• Atomic operations
  • For e.g., `float atomicAdd(float* addr, float amount)`
  • Atomic operations on both global memory and shared memory variables

• Host/device synchronization
  • Implicit barrier across all threads at return of kernel
Race Conditions and Data Races

• A race condition occurs when program behavior depends upon relative timing of two (or more) event sequences

• Execute: \( \ast c \ += \ sum; \)
  • Read value at address \( c \)
  • Add \( \sum \) to value
  • Write result to address \( c \)
Be Careful!

Thread 0, Block 0
- Read value at address c
- Add sum to value
- Write result to address c

Thread 3, Block 7
- Read value at address c
- Add sum to value
- Write result to address c
Atomic Operations

• Many read-modify-write atomic operations on memory available with CUDA C
  • atomicAdd(), atomicSub(), atomicMin(), atomicMax(), atomicInc(), atomicDec(), atomicExch(), atomicCAS()

• Predictable result when simultaneous access to memory required
References

- D. Kirk and W. Hwu – Programming Massively Parallel Processors.