Multicore Processors and GPUs: Programming Models and Compiler Optimizations

J. “Ram” Ramanujam
Louisiana State University

P. “Saday” Sadayappan
The Ohio State University

http://www.ece.lsu.edu/jxr/pact12tut/
Tutorial Outline

• Technology Trends => Multi-core Ubiquity

• Parallel Programming Models

• Introduction to Compiler Optimizations

• Optimizations for General-Purpose Multicore Architectures

• Compiler Optimizations for GPUs
The Good Old Days for Software

Source: J. Birnbaum

- Single-processor performance experienced dramatic improvements from **clock**, and **architectural** improvement (Pipelining, Instruction-Level-Parallelism)
- Applications experienced automatic performance improvement
Power Density Trends

5-year projection: 200W total, 125 W/cm²!

Power doubles every 4 years

P=VI: 75W @ 1.5V = 50 A!

• Processor clock rates have increased over 100x in two decades, but have now flattened out
Single-Processor Performance Trends

- Single-processor performance improved by about 50% per year for almost two decades, but has now flattened out: limited further upside from clock or ILP

Moore’s Law: Still Going Strong

- But transistor density continues to rise unabated
- Multiple cores are now the best option for sustained performance growth

Source: Saman Amarasinghe
The Only Option: Many Cores

- Chip density is continuing increase ~2x every 2 years
  - Clock speed is not
  - Number of processor cores may double
- There is little or no more hidden parallelism (ILP) to be found
- **Parallelism must be exposed to and managed by software**

Source: Intel, Microsoft (Sutter) and Stanford (Olukotun, Hammond)
GPU vs. CPU Performance

Source: NVIDIA

![Chart showing GPU vs. CPU performance](image-url)
GPU vs. CPU Performance

Source: NVIDIA

Energy difference: 200pJ/instruction (GPU) vs. 2000pJ/instruction (CPU)
CPU vs. GPU

• GPU’s higher performance and energy efficiency due to different allocation of chip area
  – High degree of SIMD parallelism, simple in-order cores, less control/sync. logic, lower cache/scratchpad capacity
• But SIMD parallelism is not well suited for all algorithms
  – Dynamic search, Sparse computations, Quad-tree/Oct-tree data structures, Graph algorithms, etc.
• Future systems will be heterogeneous

(Source: Nvidia CUDA Programming Guide)
Intel’s Vision of Architecture Evolution

Large, Scalar cores for high single-thread performance

Multi-core array
• CMP with ~10 cores

Scalar plus many core for highly threaded workloads

Many-core array
• CMP with 10s-100s low power cores
  • Scalar cores
  • Capable of TFLOPS+
  • Full System-on-Chip
  • Servers, workstations, embedded…

Dual core
• Symmetric multithreading

Evolution

Presentation: Paul Petersen, Sr. Principal Engineer, Intel

Tutorial Outline

• Technology Trends => Multi-core Ubiquity

• Parallel Programming Models

• Introduction to Compiler Optimizations

• Optimizations for General-Purpose Multicore Architectures

• Compiler Optimizations for GPUs
Multi-Threaded Programming Models

- Parallel program is comprised of multiple concurrent threads of computation
- Work is partitioned amongst the threads
- Data communication between the threads via shared memory or messages
  - Shared memory: More convenient than explicit messages, but danger of race conditions
  - Message passing: More tedious than use of shared memory, but lower likelihood of races
  - Examples: OpenMP, MPI, CUDA
OpenMP

- Open standard for Multi-Processing
- Standard, portable API for shared-memory multiprocessors
- Fork-Join model of parallelism; program alternates between sequential execution by “master” thread and parallel regions executed by many threads
- Primarily directive (pragma) based specification of parallelism
OpenMP: Pi Program

```c
#include <omp.h>

int num_steps = 100000;  double step;
#define N_THRDS 2
void main ()
{
    double x, step, pi, partial_sum[N_THRDS];
    step = 1.0/num_steps;
    omp_set_num_threads(N_THRDS)
    #pragma omp parallel
    {
        double x,s; int i,id;
        id = omp_get_thread_num();
        for (i=id, s=0.0; i< num_steps;i=i+N_THRDS)
            { x = (i+0.5)*step; s += 4.0/(1.0+x*x); } // Corrected the denominator
        partial_sum[id] = s;
    }

    for(i=0, pi=0.0;i<N_THRDS;i++)
        pi += partial_sum[i] * step;
}
```
OpenMP: Pi Program (version 2)

```c
#include <omp.h>
static long num_steps = 100000; double step;
define N_THRDS 2
void main ()
{
  int i; double x, pi, partial_sum[N_THRDS];
  step = 1.0/num_steps;
  omp_set_num_threads(NUM_THREADS)
  #pragma omp parallel
  {
    double s,x; int id;
    id = omp_get_thread_num();
    s = 0;
    #pragma omp for
    for (i=0;i< num_steps; i++)
      { x = (i+0.5)*step; s += 4.0/(1.0+x*x); }
    partial_sum[id] = s;
  }
  for(i=0, pi=0.0;i<N_THRDS; i++ ) pi += partial_sum[i] * step;
}
```
NVIDIA GPU Architecture

Host

CPU

Host memory

Device

Multiprocessor N

Multiprocessor 2

Multiprocessor 1

Shared Memory

Registers

Processor 1

Processor 2

Processor M

Instruction Unit

Device memory
The CUDA Model

Serial code executes in a Host (CPU) thread
Parallel code executes in many Device (GPU) threads across multiple processing elements

Source: Cliff Woolley, Nvidia
Threads

- Threads are organized into a three-level hierarchy
- Each kernel creates a single *grid* that consists of many *thread blocks* each containing many *threads*
- Threads in the same *thread block* can cooperate with each other
  - share data through the on-chip shared memory
  - can synchronize their execution using the `__syncthreads` primitive
- Threads are otherwise independent; no direct support for synchronization between threads in different thread blocks
- Threads and blocks have IDs: allows a thread to decide what data to compute on and simplifies memory addressing for multidimensional data
  - BlockID can be one or two dimensional (`blockIdx.x, blockIdx.y`)
  - ThreadID can be 1-, 2- or 3-dimensional (`threadIdx.x, threadIdx.y, threadIdx.z`)
- Threads within a block are organized into *warps* of 32 threads
- Each warp executes in SIMD fashion, issuing in one to four cycles, depending on the number of cores on each SM (Streaming Multiprocessor)
Threads and Memory Spaces

Each thread can access:

- per-thread registers
- per-block shared memory
- per-grid global memory
CUDA Extensions to C

• API along with minimal extensions to C
• Declaration specs
  - _device, _global, _shared, _local, _constant
• Special Variables
  - blockIdx, threadIdx
• Intrinsics
  - __syncthreads
• Runtime API
  - cudaMalloc(...), cudaMemcpy(...)
• Kernel execution
## CUDA Function Declarations

<table>
<thead>
<tr>
<th></th>
<th>Executed on the:</th>
<th>Only callable from the:</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__** defines a kernel function
  - Must return void
- **__device__** and **__host__** can be used together
- **__device__** functions cannot have their address taken
CUDA Variable Type Qualifiers

<table>
<thead>
<tr>
<th>Qualifiers</th>
<th>Memory</th>
<th>Scope</th>
<th>Lifetime</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>device</strong> <strong>local</strong> int</td>
<td>local</td>
<td>thread</td>
<td>thread</td>
</tr>
<tr>
<td><strong>device</strong> <strong>shared</strong> int</td>
<td>shared</td>
<td>block</td>
<td>block</td>
</tr>
<tr>
<td><strong>device</strong> int</td>
<td>global</td>
<td>grid</td>
<td>application</td>
</tr>
</tbody>
</table>

- __device__ is optional when used with __local__ or __shared__

- 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
CUDA Programming Steps

1. Allocate host memory
2. Initialize host memory
3. Allocate device memory
4. Copy host memory to device
5. Setup execution parameters
6. Execute the kernel
7. Copy result from device to host
8. Clean up memory
CUDA Programming

Matrix-matrix Multiply Example

// allocate host memory
unsigned int size_A = WA * HA;
unsigned int mem_size_A = sizeof(float) * size_A;
float* h_A = (float*) malloc(mem_size_A);
unsigned int size_B = WB * HB;
unsigned int mem_size_B = sizeof(float) * size_B;
float* h_B = (float*) malloc(mem_size_B);
unsigned int size_C = WC * HC;
unsigned int mem_size_C = sizeof(float) * size_C;
float* h_C = (float*) malloc(mem_size_C);

// initialize host memory
randomInit(h_A, size_A);
randomInit(h_B, size_B);
CUDA Programming (contd.)

// allocate device memory
float* d_A;
cudaMalloc((void**) &d_A, mem_size_A);
float* d_B;
cudaMalloc((void**) &d_B, mem_size_B);
float* d_C;
cudaMalloc((void**) &d_C, mem_size_C);

// copy host memory to device

cudAMemcpy(d_A, h_A, mem_size_A, cudaMemcpyHostToDevice);

cudAMemcpy(d_B, h_B, mem_size_B, cudaMemcpyHostToDevice);
CUDA Programming (contd.)

// setup execution parameters

dim3 threads(BLOCK_SIZE, BLOCK_SIZE);
dim3 grid(WC / threads.x, HC / threads.y);

// execute the kernel

matrixMul<<< grid, threads >>>(d_C, d_A, d_B, WA, WB);

// copy result from device to host

cudaMemcpy(h_C, d_C, mem_size_C, cudaMemcpyDeviceToHost);

// clean up memory

free(h_A);
free(h_B);
free(h_C);
cudaFree(d_A);
cudaFree(d_B);
cudaFree(d_C);
Matrix Multiplication Example

Each thread block computes one sub-matrix $C_{\text{sub}}$ of $C$. Each thread within the block computes one element of $C_{\text{sub}}$. 
MM Example (contd.)

// Tiling at thread block level
FORALL iT = 1, Ni, Ti
  FORALL jT = 1, Nj, Tj

// Tiling at thread level
FORALL it = iT, min(iT+Ti-1, Ni), Ti/ti
  FORALL jt = jT, min(jT+Tj-1, Nj), Tj/tj

  FOR kt = 1, Nk, t
    FOR i = it, min(it+Ti/ti-1, Ni)
      FOR j = jt, min(jt+Tj/tj-1, Nj)
        FOR k = kt, min(kt+t-1, Nk)
          C[i,j] += A[i,k] \* B[k,j];
          END FOR
        END FOR
      END FOR
    END FOR
END FORALL
END FORALL
END FORALL
END FORALL
END FORALL
END FORALL
END FORALL
Kernel Code

```c
__global__ void matrixMul( float* C, float* A, float* B, int hA, int wA, int wB)
{
    // Block Index and Thread Index
    int bx = blockIdx.x;
    int by = blockIdx.y;
    int tx = threadIdx.x;
    int ty = threadIdx.y;

    // Starting index of A and B for the thread block
    int aBegin = wA * BLOCK_SIZE * by;
    int bBegin = BLOCK_SIZE * bx;

    float Csub = 0;

    // Declaration of shared memory array used to store the sub-matrix of A and B
    __shared__ float As[BLOCK_SIZE][BLOCK_SIZE];
    __shared__ float Bs[BLOCK_SIZE][BLOCK_SIZE];

    // Starting index of A and B for the thread
    int alInd = aBegin + wA * ty + tx;
    int blInd = bBegin + wB * ty + tx;
```
CUDA Programming (contd.)

// Main thread-level MM kernel code

for (int kt = 0; kt < wA; kt += BLOCK_SIZE)
    As[ty][tx] = A[alnd];
    Bs[ty][tx] = B[blnd];
    __syncthreads();

    for (int k = 0; k < BLOCK_SIZE; ++k)
        Csub += As[ty][k] * Bs[k][tx];
    __syncthreads();

    alnd += BLOCK_SIZE;
    blnd += BLOCK_SIZE * wB;
}

// Copy back the result computed (from thread local memory) to GPU DRAM
int c = wB * BLOCK_SIZE * by + BLOCK_SIZE * bx;
C[c + wB * ty + tx] = Csub;
MatMul Variants


–Kernels available at: (http://www.pgroup.com/lit/kernels/kernels.tar)

–Matrices 4096 x 4096

–NVCC compiler version 2.0 run with –O option; the driver routine compiled with pgcc –fast

–Results on the GTX 280 (30 multiprocessors or SMs)

–On the GTX 280, each thread block supports up to 16 warps (16x32 scalar threads). Each SM can support up to 8 thread blocks at a time.
MatMul Variants

```c
/* SIMD-width (# threads) 32 */
mmkernel( float* a, float* b, float* c,
         int pitch_a, int pitch_b, int pitch_c,
         int n, int m, int p )
{
    int i = blockIdx.x*32 + threadIdx.x;
    int j = blockIdx.y;

    float sum = 0.0;
    for( int k = 0; k < p; ++k )
        sum += b[i+pitch_b*k] * c[k+pitch_c*j];
    a[i+pitch_a*j] = sum; }
```

k1-32.cu
MatMul Variants

/* SIMD-width (# threads) 64 */

mmkernel( float* a, float* b, float* c,
          int pitch_a, int pitch_b, int pitch_c,
          int n, int m, int p )
{
    int i = blockIdx.x*64 + threadIdx.x;
    int j = blockIdx.y;

    float sum = 0.0;
    for( int k = 0; k < p; ++k )
        sum += b[i+pitch_b*k] * c[k+pitch_c*j];
    a[i+pitch_a*j] = sum; }

k1-64.cu
MatMul Variants

**k1-32.cu**

```c
/* SIMD-width (# threads) 32 */
mmkernel( float* a, float* b, float* c,
    int pitch_a, int pitch_b, int pitch_c,
    int n, int m, int p )
{
    int i = blockIdx.x*32 + threadIdx.x;
    int j = blockIdx.y;

    float sum = 0.0;
    for( int k = 0; k < p; ++k )
        sum += b[i+pitch_b*k] * c[k+pitch_c*j];
    a[i+pitch_a*j] = sum;
}
```

**k1-64.cu**

```c
/* SIMD-width (# threads) 64 */
mmkernel( float* a, float* b, float* c,
    int pitch_a, int pitch_b, int pitch_c,
    int n, int m, int p )
{
    int i = blockIdx.x*64 + threadIdx.x;
    int j = blockIdx.y;

    float sum = 0.0;
    for( int k = 0; k < p; ++k )
        sum += b[i+pitch_b*k] * c[k+pitch_c*j];
    a[i+pitch_a*j] = sum;
}
```

<table>
<thead>
<tr>
<th>Version</th>
<th>k1-32</th>
<th>k1-64</th>
<th>k1-128</th>
<th>k1-256</th>
</tr>
</thead>
<tbody>
<tr>
<td>GFLOPs</td>
<td>28</td>
<td>36</td>
<td>35</td>
<td>35</td>
</tr>
</tbody>
</table>

Louisiana State University  The Ohio State University
MatMul Variants

/* SIMD-width (# threads) 32 [1x32] thread block */
extern "C" __global__ void
mmkernel( float* a, float* b, float* c,
    int pitch_a, int pitch_b, int pitch_c,
    int n, int m, int p )
{
    int i = blockIdx.x*32 + threadIdx.x;
    int j = blockIdx.y;

    float sum = 0.0;
    for( int k = 0; k < p; ++k )
        sum += b[i+pitch_b*k] * c[k+pitch_c*j];
    a[i+pitch_a*j] = sum;  }

k1-32.cu
MatMul Variants

/* SIMD-width (# threads) 32 [32x1] thread block */
extern "C" __global__ void 
mmkernel( float* a, float* b, float* c,
    int pitch_a, int pitch_b, int pitch_c,
    int n, int m, int p )
{
    int i = blockIdx.y;
    int j = blockIdx.x*32 + threadIdx.x;

    float sum = 0.0;
    for( int k = 0; k < p; ++k )
        sum += b[i+pitch_b*k] * c[k+pitch_c*j];
    a[i+pitch_a*j] = sum;   }

k1r-32.cu: Non-stride-1 array access
## MatMul Variants

```c
/* SIMD-width (# threads) 32 [1x32] thread block */
extern "C" __global__ void
mmkernel( float* a, float* b, float* c,
        int pitch_a, int pitch_b, int pitch_c,
        int n, int m, int p )
{
    int i = blockIdx.x*32 + threadIdx.x;
    int j = blockIdx.y;

    float sum = 0.0;
    for( int k = 0; k < p; ++k )
        sum += b[i+pitch_b*k] * c[k+pitch_c*j];
    a[i+pitch_a*j] = sum;
}
```

```c
/* SIMD-width (# threads) 32 [32x1] thread block */
extern "C" __global__ void
mmkernel( float* a, float* b, float* c,
        int pitch_a, int pitch_b, int pitch_c,
        int n, int m, int p )
{
    int i = blockIdx.y;
    int j = blockIdx.x*32 + threadIdx.x;

    float sum = 0.0;
    for( int k = 0; k < p; ++k )
        sum += b[i+pitch_b*k] * c[k+pitch_c*j];
    a[i+pitch_a*j] = sum;
}
```

<table>
<thead>
<tr>
<th>Version</th>
<th>k1-32</th>
<th>k1r-32</th>
</tr>
</thead>
<tbody>
<tr>
<td>GFLOPs</td>
<td>28</td>
<td>1.7</td>
</tr>
</tbody>
</table>

---

Louisiana State University  The Ohio State University
MatMul Variants

/* SIMD-width (# threads) 32 [1x32] thread block */

mmkern( float* a, float* b, float* c,
    int pitch_a, int pitch_b, int pitch_c,
    int n, int m, int p )
{
    int i = blockIdx.x*32 + threadIdx.x;
    int j = blockIdx.y;
    float sum = 0.0;
    for( int k = 0; k < p; ++k )
        sum += b[i+pitch_b*k] * c[k+pitch_c*j];
    a[i+pitch_a*j] = sum;
}

/* matmul, strip mine k loop, use shared memory */

mmkern( float* a, float* b, float* c,
    int pitch_a, int pitch_b, int pitch_c,
    int n, int m, int p )
{
    int tx = threadIdx.x;
    int i = blockIdx.x*32 + tx;
    int j = blockIdx.y;
    __shared__ float cb[32];
    float sum = 0.0;
    for( int ks = 0; ks < p; ks += 32 ){
        cb[tx] = c[ks+tx+pitch_c*j];
        __syncthreads();
        for( int k = ks; k < ks+32; ++k )
            sum += b[i+pitch_b*k] * cb[k-ks];
    }
    a[i+pitch_a*j] = sum;
}

<table>
<thead>
<tr>
<th>Version</th>
<th>k1</th>
<th>k2</th>
<th>k2-64</th>
<th>k2-128</th>
<th>k2-256</th>
</tr>
</thead>
<tbody>
<tr>
<td>GFLOPs</td>
<td>28</td>
<td>33</td>
<td>55</td>
<td>63</td>
<td>63</td>
</tr>
</tbody>
</table>

k2.cu (uses shared mem)

Louisiana State University

The Ohio State University
/* strip mine k loop, use shared-mem, unroll i loop */

mmkernel( float* a, float* b, float* c,
        int pitch_a, int pitch_b, int pitch_c,
        int n, int m, int p)
{
    int tx = threadIdx.x;
    int i = blockIdx.x*64 + tx;
    int j = blockIdx.y;
    __shared__ float cb[32];

    float sum0 = 0.0, sum1 = 0.0;
    for( int ks = 0; ks < p; ks += 32 ){
        cb[tx] = c[ks+tx+pitch_c*j];
        __syncthreads();
        for( int k = ks; k < ks+32; ++k ){
            sum0 += b[i+pitch_b*k] * cb[k-ks];
            sum1 += b[i+32+pitch_b*k] * cb[k-ks];
        }
        __syncthreads();
    }
    a[i+pitch_a*j] = sum0;
    a[i+32+pitch_a*j] = sum1;
}

/* strip mine k loop, use shared-mem, unroll j loop */

mmkernel( float* a, float* b, float* c,
        int pitch_a, int pitch_b, int pitch_c,
        int n, int m, int p)
{
    int tx = threadIdx.x;
    int i = blockIdx.x*32 + tx;
    int j = blockIdx.y*2;
    __shared__ float cb0[32], cb1[32];

    float sum0 = 0.0, sum1 = 0.0;
    for( int ks = 0; ks < p; ks += 32 ){
        cb0[tx] = c[ks+tx+pitch_c*j];
        cb1[tx] = c[ks+tx+pitch_c*(j+1)];
        __syncthreads();
        for( int k = ks; k < ks+32; ++k ){
            float rb = b[i+pitch_b*k];
            sum0 += rb * cb0[k-ks];
            sum1 += rb * cb1[k-ks];
        }
        __syncthreads();
    }
    a[i+pitch_a*j] = sum0;
    a[i+32+pitch_a*(j+1)] = sum1;
}
MatMul Variants

Base (k1): SIMD width 32/64/128/256: 28/36/35/35 GFLOPs

Use Shared-Mem (k2): 32/64/128/256: 33/55/63/63 GFLOPs

Use SharedMem; Unroll i (k3): 32/64: 53/63 GFLOPs

Use ShMem; Unroll j (k4): 32/64/128: 59/98/117 GFLOPs

Use ShMem; 3x-unroll j & k; SIMD-width 128: 208 GFLOPs
Lessons from MatMul Variants

- The performance of the GPU is highly sensitive to the formulation of the kernel; needs significant experimentation
- Programmers may like this low level of control and this is appropriate for library development
- But, this is true for the CPU as well; but compilers and tools for CPU are far more mature and helpful
- Compiler optimizations are being developed for GPUs
OpenCL: Open Computing Language

- New language (based on C99) standard for portable programming of heterogeneous systems with CPUs, GPUs, and other processors
- Developed by an industry consortium
  - AMD, Apple, ARM, Broadcom, Codeplay, Electronic Arts, Ericsson, Freescale, IBM, Intel, Nokia, NVIDIA, Qualcomm, RapidMind, Sony, TI,
- Initial implementations had severe performance bottlenecks (compared to CUDA), but significant improvements of late
  - Several implementors today: AMD, Apple, ARM, Creative Labs, IBM, Imagination Technologies, Intel, NVIDIA, ST Microelectronics, Vivante
OpenCL Platform Model

- **Host (CPU)** connected to one or more **Compute Devices** (Accelerators)
  - **Kernels** queued for execution on Compute Devices
- Two-level parallel structure
  - Compute Device is comprised of one or more **Compute Units**
  - Compute Unit has one or more **Processing Elements**
OpenCL Execution Model

• **Compute Kernel**
  – Basic unit of executable code for compute devices
  – **Two models:** Data-Parallel and Task-Parallel

• **Host Program**
  – Runs on host and manages the execution of compute kernels on compute devices

• **Data Parallel Kernel Execution**
  – Kernel invoked over an index space NDRange (1-3D)
  – Work-item (equiv. CUDA thread) is an instance of kernel executed at each point in the index space
  – Work-items can be grouped together into work-groups (equiv. CUDA thread-block); synchronization only within work-group
Data Parallel Execution Model

\[(g_x, g_y) = (w_x \cdot S_x + s_x, w_y \cdot S_y + s_y)\]

Two-level parallel view:
- Each work-item has a global-id with NDRange components, e.g. \((g_x, g_y)\)
- A work item also has a local-id \((s_x, s_y)\) within its work-group
Task Parallel Model

• Single instance of kernel is executed independent of any index-space

• Logically equivalent to executing kernel on a compute unit with a work-group containing a single work-item

• Parallelism exploited by:
  – using vector data types implemented by the device,
  – enqueueing multiple tasks, and/or
  – enqueueing native kernels
OpenCL Memory Model

- Multiple distinct addr. spaces
  - Global (accessible by all work-items in all work-groups; may be cached, depending on device capability)
  - Local (accessible by work-items within a single work-group)
  - Private (only visible to a single work-item)
- Relaxed consistency model
  - Load/store consistency within single work-item
  - Guaranteed consistency between work-items of a work-group (for local memory or global memory) at work-group barrier
  - No consistency guarantees between work-items in different work groups of a kernel

Source: A. Munshi

---

Louisiana State University

The Ohio State University
Context and Command Queues

- **Host program creates and uses contexts:**
  - Devices: The collection of OpenCL devices to be used by the host
  - Kernels: The OpenCL functions that run on OpenCL devices
  - Program Objects: The program source and executable that implement the kernels
  - Memory Objects: A set of memory objects visible to the host and the OpenCL devices

- **Host program creates/uses command queues**
  - Launch kernel execution in context
  - Orchestrate data movement between memory objects
  - Synchronize between commands in queue
Host Program Structure

• Query compute devices
• Create contexts
  – Create memory objects associated with contexts
  – Compile and create kernel program objects
• Issue commands to command-queue
  – Kernel execution
  – Data copy
• Synchronization of commands
• Free OpenCL resources
Example: Kernel Function

```c
__kernel void vec_add (__global const float *a, 
__global const float *b, __global float *c)
{
    int gid = get_global_id(0);
    c[gid] = a[gid] + b[gid];
}
```

Source: J. Yang
Example: Host Program (Part 1)

// create the OpenCL context on a GPU device
cl_context cxt =
clCreateContextFromType(0,CL_DEVICE_TYPE_GPU,NULL, NULL, NULL);
// get the list of GPU devices associated with context
cGetContextInfo(cxt, CL_CONTEXT_DEVICES, 0, NULL, &cb);
cl_device_id *devices = malloc(cb);
cGetContextInfo(cxt, CL_CONTEXT_DEVICES, cb, devices, NULL);

// allocate the buffer memory objects
cl_mem memobjs[3];
memobjs[0] = clCreateBuffer(cxt, CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR, sizeof(cl_float)*n, srcA, NULL);
memobjs[1] = clCreateBuffer(cxt,CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR, sizeof(cl_float)*n, srcB, NULL);
memobjs[2] = clCreateBuffer(cxt, CL_MEM_WRITE_ONLY, sizeof(cl_float)*n, NULL, NULL);
Example: Host Program (Part 2)

```c
// create a command-queue
cl_cmd_queue cq = clCreateCommandQueue(cxt, devices[0], 0, NULL);

// create the program
cl_program pgm = clCreateProgramWithSource(cxt, 1, &program_source, NULL, NULL);

// build the program
cl_int err = clBuildProgram(pgm, 0, NULL, NULL, NULL, NULL, NULL);

// create the kernel
cl_kernel kl = clCreateKernel(pgm, "vec_add", NULL);

// set the args values
err  = clSetKernelArg(kl, 0, (void *)&memobjs[0], sizeof(cl_mem));
err  = clSetKernelArg(kl, 1, (void *)&memobjs[1], sizeof(cl_mem));
err  = clSetKernelArg(kl, 2, (void *)&memobjs[2], sizeof(cl_mem));
```
Example: Host Program (Part 3)

// set work-item dimensions
size_t glwksz[0] = n;

// execute kernel
err = clEnqueueNDRangeKernel(cq, kl, 1, NULL, glwksz, NULL, 0, NULL, NULL);

// read output array
err = clEnqueueReadBuffer(cxt, memobjs[2], CL_TRUE, 0, n*sizeof(cl_float), dst, 0, NULL, NULL);
PGI Accelerator

• Directive based approach for automatically transforming sequential C/Fortran programs by pgcc compiler for execution on GPUs

• Very similar approach to OpenMP
  – Compiler transforms code within “acc” region into kernel code and loads on GPU
  – All memory allocation in GPU and data transfers between host and GPU managed by compiler
  – User can optionally guide to compiler via pragma clauses

• Influenced many features of new OpenACC standard
Accelerator Directives

```c
change = tolerance + 1.0
!

do while (change > tolerance)
    change = 0
!

do i = 2, m-1
    do j = 2, n-1
        newa(i,j) = w0 * a(i,j) + 
        w1 * (a(i-1,j) + a(i,j-1) + &
        a(i+1,j) + a(i,j+1)) + &
        w2 * (a(i-1,j-1) + a(i-1,j+1) + &
        a(i+1,j-1) + a(i+1,j+1))
        change = max(change, abs(newa(i,j) - a(i,j)))
    enddo
endo
enddo

a(2:m-1,2:n-1) = newa(2:m-1,2:n-1)
!

do j = 2, n-1
    do i = 2, m-1
        newa(i,j) = w0 * a(i,j) + &
        w1 * (a(i-1,j) + a(i,j-1) + &
        a(i+1,j) + a(i,j+1)) + &
        w2 * (a(i-1,j-1) + a(i-1,j+1) + &
        a(i+1,j-1) + a(i+1,j+1))
        change = max(change, abs(newa(i,j) - a(i,j)))
endo
endo

a(2:m-1,2:n-1) = newa(2:m-1,2:n-1)
!

Example Source: D. Miles (PGI)
```
Accelerator Directives

Automatic versus User-specified mapping

Example Source: D. Miles (PGI)
void saxpy (float a,
  float *restrict x,
  float *restrict y, int n){
    #pragma acc region
    for (int i=1; i<n; i++)
      x[i] = a*x[i] + y[i];
}

PGI Accelerator Compilers

Host x86 Code

saxpy:
...
  movl (%rbx), %eax
  movl %eax, -4(%rbp)
  call __pg_cu_init
  ...
  call __pg_cu_alloc
  ...
  call __pg_cu_uploadp
  ...
  call __pg_cu_paramset
  ...
  call __pg_cu_launch
  ...
  Call __pg_cu_downloadp
...

static __constant__ struct{
  int tc1;
  float* _y;
  float* _x;
  float* _a;
  }a2;
extern "C" __global__ void
gpu_kernel_2() {
  int i1, i1s, ibx, itx;
  ibx = blockIdx.x;
  itx = threadIdx.x;
  for( i1s = ibx*256; i1s < a2.tc1; i1s += gridDim.x*256 ){
    i1 = itx + i1s;
    if( i1 < a2.tc1 ){
      a2._x[i1] = (a2._y[i1]+(a2._x[i1]*a2._a));
    }
  }
}

... with no change to existing makefiles, scripts, programming environment, etc

Source: D. Miles (PGI)
PGI Accelerator

• **Key compiler steps**
  
  – Determine what **computations** (done by the programmer using directives to indicate accelerator compute regions) and hence what **data**, should be moved **to the accelerator** (done by the compiler using data flow, alias and array regions analysis).

  – **The Planner** - Determine how the **loop parallelism** will **map** onto the **hardware parallelism**: Translating from loop indices into **blockIdx** and **threadIdx** indices or global and local thread indices.

  – Generate and optimize the kernel code: As CUDA programmers have learned, manual code tuning can make a significant improvement in performance.
!$acc data region copy(a(1:n,1:m)) &
!$acc& local(b(2:n-1,2:m-1)) copyin(w(2:n-1)
  do while(resid.gt.tol)
    resid = 0.0
!$acc region
  do i = 2, n-1
    do j = 2, m-1
      b(i,j) = 0.25*w(i)*(a(i-1,j)+a(i,j-1)+ a(i+1,j)+a(i,j+1))
        +(1.0-w(i))*a(i,j)
    enddo
  enddo
  do i = 2, n-1
    do j = 2, m-1
      resid = resid + (b(i,j)-a(i,j))**2
      a(i,j) = b(i,j)
    enddo
  enddo
!$acc end region
enddo
!$acc end data region
!$acc data region copy(a(1:n,1:m)) &
!$acc& local(b(2:n-1,2:m-1)) copyin(w(2:n-1)

do while(resid.gt.tol)
    resid = 0.0
!$acc region
    do i = 2, n-1
        do j = 2, m-1
            b(i,j) = 0.25*w(i)*(a(i-1,j)+a(i,j-1)+ a(i+1,j)+a(i,j+1))
            +(1.0-w(i))*a(i,j)
        enddo
    enddo
    do i = 2, n-1
        do j = 2, m-1
            resid = resid + (b(i,j)-a(i,j))^2
            a(i,j) = b(i,j)
        enddo
    enddo
!$acc end region
enddo
!$acc end data region
PGI Accelerator – Planner

- Applies multiple levels of tiling and map each loop to one of the following:
  - to the block parallel dimension,
  - to the thread parallel dimension,
  - to sequential execution within a thread, or
  - to be unrolled within a thread, which is essentially the same as sequential execution.

```plaintext
do i0 = 2, n-1, bi0
  do j0 = 2, m-1, bj0
    do il = i0, min(n-1,i0+bi0-1), bi1
      do j1 = j0, min(m-1,j0+bj0-1), bj1
        do i = il, min(n-1,il+bi1-1)
          do j = j1, min(m-1,j1+bj1-1)
            b(i,j) = ...
```
PGI Accelerator – Planner

\[
\begin{align*}
\text{do } i_0 &= 2, \ n-1, \ b_i0 \\
\text{do } j_0 &= 2, \ m-1, \ b_j0 \\
&\quad \text{do } i_1 = i_0, \ \min(n-1,i_0+b_i0-1), \ b_i1 \\
&\quad \quad \text{do } j_1 = j_0, \ \min(m-1,j_0+b_j0-1), \ b_j1 \\
&\quad \quad \quad \text{do } i = i_1, \ \min(n-1,i_1+b_i1-1) \\
&\quad \quad \quad \quad \text{do } j = j_1, \ \min(m-1,j_1+b_j1-1) \\
&\quad \quad \quad \quad \ b(i,j) = \ldots
\end{align*}
\]

- $i_0/j_0$ loops -> block parallelism; $b_i0$ and $b_j0$ determine the size of the grid blocks
- $i/j$ loops -> thread parallelism
- The $i_1/j_1$ (middle) loops -> seq. exec. In a block
- Global memory coalescing: which loop maps to which dimension – does the $i$ loop map to the x-dimension or does the $j$-loop?
PGI Accelerator – Planner Strategy

• **Tile the loops twice at two levels**, and reordering the loops so the outermost tiled loops correspond to the grid indices and the innermost loops are the thread indices.

• **Order the outermost loops** (assigning grid x, y indices to loops).

• **Order the innermost loops** (assigning thread x, y, z indices to loops); assign the loop index that accesses stride-1 arrays to the thread x dimension.

• **Order the sequential intermediate loops** - often to optimize redundant data accesses (see array w in the example).

• **Choosing tile size** (the strip size for the thread index loops); the Planner tries to choose a tile size to maximize parallelism and tune for redundancy between threads.

• **Selecting which data to store in the data cache** -- the cache is limited in size, and the Planner may have to throttle parallelism to remove more redundant data accesses.
OpenACC (Source: NVIDIA)

- Directives: easy path to accelerate compute-intensive applications
- Simple compiler hints added to C or Fortran source - similar to OpenMP directives
- This works on many-core GPUs and multicore CPUs

"OpenACC will enable programmers to easily develop portable applications that maximize the performance and power efficiency benefits of the hybrid CPU/GPU architecture of Titan."

--Buddy Bland, Titan Project Director, Oak Ridge National Lab
OpenACC [http://www.openacc.org/] (source: NVIDIA)

- **Compiler directives to specify parallel regions in C and Fortran**
  - OpenACC compilers offload parallel regions from host to accelerator
  - Designed to be portable across host CPUs, accelerators, operating systems and compilers

- **Create high-level heterogeneous programs**
  - Without explicit accelerator initialization and without explicit data or program transfers between host and accelerator

- **The programming model allows programmers to start simple**
  - Enhance with additional guidance for compiler on loop mappings, data location, and other performance details

- **Compatibility with other GPU languages and libraries**
  - Enable interoperability between CUDA C/Fortran and GPU libraries such as CUBLAS, CULA, CUFFT, etc.
OpenACC – saxpy kernel

C:

```c
#pragma acc directive [clause [,] clause] …
... often followed by a structured code block

void saxpy(int n, float a, float *x, float *restrict y)
{
    #pragma acc kernels
    for (int i = 0; i < n; ++i)
        y[i] = a*x[i] + y[i];
}
...
// Perform SAXPY on one million elements
saxpy(1<<20, 2.0, x, y);
...
```

A kernel is a parallel function that executes on the GPU
kernels construct

- Each loop is executed as a separate kernel on the GPU. The **kernels** construct is derived from PGI Accelerator’s *region* construct.
Jacobi iteration

while ( err > tol && iter < iter_max ) {
    err=0.0;

    for( int j = 1; j < n-1; j++) {
        for(int i = 1; i < m-1; i++) {

            Anew[j][i] = 0.25 * (A[j][i+1] + A[j][i-1]
                                + A[j-1][i] + A[j+1][i]);

            err = max(err, abs(Anew[j][i] - A[j][i]));
        }
    }

    for( int j = 1; j < n-1; j++) {
        for( int i = 1; i < m-1; i++ ) {
            A[j][i] = Anew[j][i];
        }
    }

    iter++;
}
while ( err > tol && iter < iter_max ) {
    err=0.0;

#pragma acc kernels reduction(max:err)
    for( int j = 1; j < n-1; j++ ) {
        for(int i = 1; i < m-1; i++) {

            Anew[j][i] = 0.25 * (A[j][i+1] + A[j][i-1] 
                            + A[j-1][i] + A[j+1][i]);

            err = max(err, abs(Anew[j][i] - A[j][i]));
        }
    }

#pragma acc kernels
    for( int j = 1; j < n-1; j++ ) {
        for( int i = 1; i < m-1; i++ ) {
            A[j][i] = Anew[j][i];
        }
    }

    iter++;
}
Data Clauses

• **copy ( list )**: Allocates memory on GPU and copies data from host to GPU when entering region and copies data to the host when exiting region.

• **copyin ( list )**: Allocates memory on GPU and copies data from host to GPU when entering region.

• **copyout ( list )**: Allocates memory on GPU and copies data to the host when exiting region.

• **create ( list )**: Allocates memory on GPU but does not copy.

• **present ( list )**: Data is already present on GPU from another containing data region.
Update construct

#pragma acc update [clause ...]

if( expression )
async( expression )

• Used to update existing data after it has changed in its corresponding copy (e.g. update device copy after host copy changes)
• Move data from GPU to host, or host to GPU.
• Data movement can be conditional, and asynchronous.
Jacobi iteration

```c
#pragma acc data copy(A), create(Anew)
while ( err > tol && iter < iter_max ) {
    err=0.0;

#pragma acc kernels reduction(max:err)
    for( int j = 1; j < n-1; j++ ) {
        for(int i = 1; i < m-1; i++) {


            err = max(err, abs(Anew[j][i] - A[j][i]));
        }
    }

#pragma acc kernels
    for( int j = 1; j < n-1; j++ ) {
        for( int i = 1; i < m-1; i++ ) {
            A[j][i] = Anew[j][i];
        }
    }

    iter++;
}
```
**parallel** construct

- The entire **parallel** construct becomes a single target parallel operation; in CUDA terms, the parallel construct becomes a single CUDA kernel.
- In contrast, each loop nest in the **kernels** construct is compiled and launched separately. In CUDA terms, each loop nest becomes a separate CUDA kernel. In particular, this means that the code for the first loop nest will complete before the second loop nest begins execution.
- See [http://www.pgroup.com/lit/articles/insider/v4n2a1.htm](http://www.pgroup.com/lit/articles/insider/v4n2a1.htm) for Michael Wolfe’s detailed discussion of the **parallel** and **kernels** constructs in OpenACC
OpenACC

- OpenACC gives us more detailed control over parallelization
  - via gang, worker, and vector clauses
  - Block similar to thread, worker similar to warp, vector similar to thread

- Allows mapping to an architecture that is a collection of Processing Elements (PEs)
  - One or more PEs per node
  - Each PE is multi-threaded
  - Each thread can execute vector instructions

- By understanding more about OpenACC execution model and GPU hardware organization, one can get higher speedups

- By understanding bottlenecks in the code via profiling, one can reorganize the code for higher performance
Summary

• Dramatic change in technology trends since 2005 => increasing number of cores has become the main means to performance improvement
  – GPUs exploit significant amount of SIMD parallelism

• Software for GPUs must be highly multi-threaded
  – Explicitly parallel programming models for GPUs
    • CUDA, OpenCL, PGI-Accelerator, HMPP-CAPS, OpenACC
  – Automatic compiler transformation of sequential code
    • R-Stream compiler (Reservoir Labs), IBM XL, Pluto
  – Many compiler techniques for automatic transformation of sequential code will also be useful for OpenACC
References

• Michael Wolfe’s HPCwire articles on GPU optimization

• CUDA

• OpenCL
  – http://www.khronos.org/registry/cl/specs/opencl-1.2.pdf

• PGI Accelerator Tutorial

• OpenACC Tutorial

• David Kirk & Wen-Mei Hwu’s GPU course
  – http://impact.crhc.illinois.edu/gpucourses.php