

# GPU Programming with CUDA (C and PGI CUDA Fortran) and the PGI Accelerator Programming Model

Michael Wolfe  
[Michael.Wolfe@pgroup.com](mailto:Michael.Wolfe@pgroup.com)  
<http://www.pgroup.com>

March 2011

## Part 1: Introduction

1-1

The Portland Group®

## Agenda

- Introduction
  - GPU architecture vs. Host architecture
  - GPU parallel programming vs. Host parallel programming
- CUDA
  - CUDA Programming model
  - The Host Program
  - Writing Kernels
  - Building and Running CUDA Programs
  - Performance Tuning
- Accelerator Model
  - PGI Accelerator Programming Model
  - Building PGI Accelerator Programs
  - Directive Details
  - Interpreting Compiler Feedback
  - Tips and Tricks
  - Performance Tuning

1-2

The Portland Group®

## Host Architecture Features

- Register files (integer, float)
- Functional units (integer, float, address), lcache, Dcache
- Execution pipeline (fetch, decode, issue, execute, cache, commit)
  - branch prediction, hazards (control, data, structural)
  - pipelined functional units, superpipelining, register bypass
  - stall, scoreboarding, reservation stations, register renaming
- Multiscalar execution (superscalar, control unit lookahead)
  - LIW (long instruction word)
- Multithreading, Simultaneous multithreading
- Vector instruction set
- Multiprocessor, Multicore, coherent caches (MESI protocols)

1-3

The Portland Group®

## Example Systems

- Scalar pipeline
  - MIPS, SPARC
- LIW
  - Multiflow Trace, Cydrome Cydra-5, i860
- Multithreaded
  - Denelcor HEP, Tera, GPUs, Sparc Niagara
  - SMT: Intel hyperthreading
- Vector
  - Cray, Convex, NEC
- Multiprocessors
  - Sequent, Encore, SGI, many

1-4

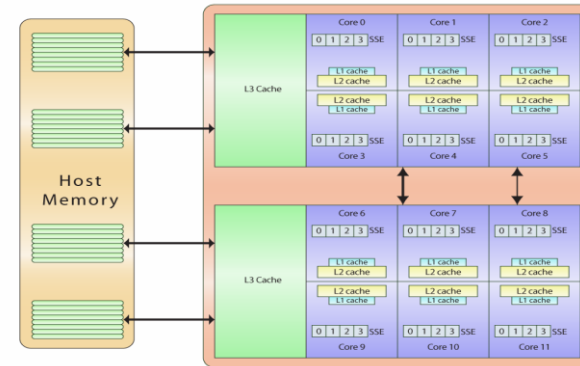
The Portland Group®

## Making It Faster

- **Processor:**
  - Faster clocks
  - More work per clock:
    - superscalar
    - VLIW
    - more cores
    - vector / SIMD instructions
- **Memory**
  - Latency reduction
  - Latency tolerance

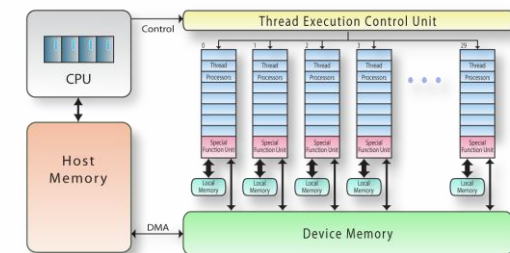
1-5

## AMD "Magny-Cours"



©2010 The Portland Group, Inc.

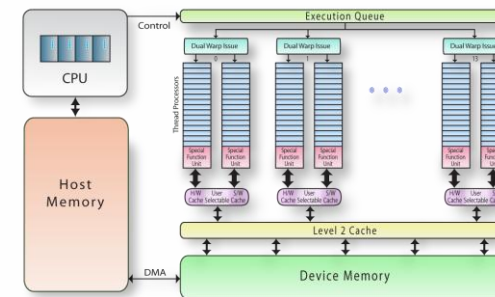
## Abstracted x64+Tesla Architecture



©2010 The Portland Group, Inc.

1-7

## Abstracted x64+Fermi Architecture



©2010 The Portland Group, Inc.

1-8

## GPU Architecture Features

- ❑ Optimized for high degree of regular parallelism
- ❑ Classically optimized for low precision
  - Fermi supports double precision at ½ single precision bandwidth
- ❑ High bandwidth memory (Fermi supports ECC)
- ❑ Highly multithreaded (slack parallelism)
- ❑ Hardware thread scheduling
- ❑ Non-coherent software-managed data caches
  - Fermi has two-level hardware data cache
- ❑ No multiprocessor memory model guarantees
  - some guarantees with fence operations

1-9



## Tesla-10 Features Summary

- ❑ Massively parallel thread processors
  - Organized into multiprocessors
    - up to 30, see deviceQuery or pgaccelinfo
  - Physically: 8 thread processors per multiprocessor
  - Logically: 32 threads per warp
- ❑ Memory hierarchy
  - host memory, device memory, constant memory, shared memory, register
- ❑ Queue of operations (kernels) on device

1-10



## Fermi (Tesla-20) Features Summary

- ❑ Massively parallel thread processors
  - Organized into multiprocessors
    - up to 16, see deviceQuery or pgaccelinfo
  - Physically: two groups of 16 thread processors per multiprocessor
  - Logically: still 32 threads per warp
- ❑ Memory hierarchy
  - host memory, device memory (two level hardware cache), constant memory, (configurable) shared memory, register
- ❑ Queue of operations (kernels) on device
- ❑ ECC memory protection (supported, not default)
- ❑ Much improved double precision performance
- ❑ Hardware 32-bit integer multiply

1-11



## Identifying your GPU

- ❑ pgaccelinfo
- ❑ deviceQuery (CUDA SDK)
- ❑ Linux
  - NVIDIA driver powers down inactive devices
- ❑ Windows
  - You must be at the console to access the GPU
- ❑ Mac OSX
  - Your notebook may have two GPUs, but OS will power one down

1-12



## Parallel Programming on CPUs

- ❑ Instruction level parallelism (ILP)
  - Loop unrolling, instruction scheduling
- ❑ Vector parallelism
  - Vectorized loops (or vector intrinsics)
- ❑ Thread level / Multiprocessor / multicore parallelism
  - Parallel loops, parallel tasks
  - Posix threads, OpenMP, Cilk, TBB, .....
- ❑ Large scale cluster / multicomputer parallelism
  - MPI (& HPF, co-array Fortran, UPC, Titanium, X10, Fortress, Chapel)

1-13

The Portland Group®

## pthread main routine

```
call pthread_create( t1, NULL, jacobi, 1 )
call pthread_create( t2, NULL, jacobi, 2 )
call pthread_create( t3, NULL, jacobi, 3 )
call jacobi( 4 )
call pthread_join( t1, NULL )
call pthread_join( t2, NULL )
call pthread_join( t3, NULL )
```

1-14

The Portland Group®

## pthread subroutine

```
subroutine jacobi( threadnum )
  lchange = 0
  do j = threadnum+1, n-1, numthreads
    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))
      lchange = max(lchange, abs(newa(i,j)-a(i,j)))
    enddo
  enddo
  call pthread_mutex_lock( lck )
  change = max(change, lchange)
  call pthread_mutex_unlock( lck )
end subroutine
```

1-15

The Portland Group®

## Jacobi Relaxation with OpenMP directives

```
change = 0
!$omp parallel private(i,j)
!$omp do reduction(max:change)
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)))
  enddo
enddo
!$omp end parallel
```

1-16

The Portland Group®

## Behind the Scenes

- **Compiler generates code for N threads:**
  - split up the iterations across N threads
  - accumulate N partial sums (no synchronization)
  - accumulate final sum as threads complete
- **Assumptions**
  - uniform memory access costs
  - coherent cache mechanism

1-17

The Portland Group®

## More Behind the Scenes

- **Virtualization penalties**
  - load balancing
  - cache locality
  - vectorization within the threads
  - thread management
  - loop scheduling (which thread does what iteration)
  - NUMA memory access penalty

1-18

The Portland Group®

## Parallel Programming on GPUs

- **High degree of regular parallelism**
  - lots of scalar threads
  - threads organized into thread groups / blocks
    - SIMD, pseudo-SIMD
  - thread groups organized into grid
    - MIMD
- **Languages**
  - CUDA, OpenCL, (Brook, Brook+), graphics: OpenGL, DirectX
  - may include vector datatypes (float4, int2)
- **Platforms**
  - (Rapidmind, now owned by Intel)

1-19

The Portland Group®

## GPU Programming

- **Allocate data on the GPU**
- **Move data from host, or initialize data on GPU**
- **Launch kernel(s)**
  - GPU driver can generate ISA code at runtime
  - preserves forward compatibility without requiring ISA compatibility
- **Gather results from GPU**
- **Deallocate data**

1-20

The Portland Group®

## Appropriate GPU programs

- ❑ Characterized by nested parallel loops
- ❑ High compute intensity
- ❑ Regular data access
- ❑ Isolated host/GPU data movement

1-21

The Portland Group®

## Host-side CUDA C GPU Control Code

```
__device__ float change;

memsize = sizeof(float)*n*m
cudaMalloc( &da, memsize );
cudaMalloc( &dnewa, memsize );
cudaMalloc( &lchange, (n/16)*(m/16) );

cudaMemcpy( da, a, memsize, cudaMemcpyHostToDevice );

dim3 threads( 16, 16 );
dim3 blocks( n/16, m/16 );
jacobikernel<<<blocks,threads>>>( da, dnewa, lchange, n, m );
reductionkernel<<<1,256>>>( lchange, (n/16)*(m/16) );

cudaMemcpy( a, dnewa, memsize, cudaMemcpyDeviceToHost );

cudaFree( da );
cudaFree( dnewa );
cudaFree( lchange );
```

1-23

The Portland Group®

## Jacobi Relaxation

```
change = 0;
for( i = 1; i < m-1; ++i ){
  for( j = 1; j < n-1; ++j ){
    newa[j][i] = w0*a[j][i] +
      w1 * (a[j][i-1] + a[j-1][i] +
            a[j][i+1] + a[j+1][i]) +
      w2 * (a[j-1][i-1] + a[j+1][i-1] +
            a[j-1][i+1] + a[j+1][i+1]);
    change = fmaxf( change, fabsf( newa[j][i] - a[j][i] ) );
  }
}
```

1-22

The Portland Group®

## Device-side CUDA C (1)

```
extern "C" __global__ void
jacobikernel( float* a, float* anew, float* lchange, int n, int m )
{
  int ti = threadIdx.x, tj = threadIdx.y; /* local indices */
  int i = blockIdx.x*16+ti+1, j = blockIdx.y*16+tj+1; /* global */
  __shared__ float mychange[16*16];
  float mya, oldmya = a[j*m+i];

  mya = w0 * oldmya +
    w1 * (a[j*m+i-1] + a[(j-1)*m+i] +
          a[j*m+i+1] + a[(j+1)*m+i] +
          w2 * (a[(j-1)*m+i-1] + a[(j+1)*m+i-1] +
                a[(j-1)*m+i+1] + a[(j+1)*m+i+1]);
  anew[j*m+i] = mya;
  /* this thread's "change" */
  mychange[ti+16*tj] = fabs(mya-oldmya);
  __syncthreads();
}
```

1-24

The Portland Group®

## Device-side CUDA C (2)

```
/* reduce all "change" values for this thread block
 * to a single value */
n = 256;
while( (n >>= 1) > 0 ){
    if( tx+ty*16 < n )
        mychange[ti+tj*16] = fmaxf( mychange[ti+tj*16],
                                    mychange[ti+tj*16+n]);
    __syncthreads();
}
/* store this thread block's "change" */
if(tx==0&&ty==0)
    lchange[blockIdx.x*gridDim.x*blockIdx.y] = mychange[0];
}
```

1-25



## Device-side CUDA C (3)

```
/* reduce all thread block's "change" values to a single value */
extern "C" __global void
reductionkernel( float* lchange, int n )
{
    __shared__ float mychange[256];
    float mych;
    int i = threadIdx.x, m;
    mych = lchange[i];
    m = 256;
    while( m <= n ){
        mych = fmaxf(mych, lchange[m]);
        m += 256;
    }
    mychange[i] = mych;
    __syncthreads();
}
```

1-26



## Device-side CUDA C (4)

```
n = 256;
while( (n >>= 1) > 0 ){
    if(i<n) mychange[i] = fmaxf(mychange[i],mychange[i+n]);
    __syncthreads();
}
if(i==0) lchange[0] = mychange[0];
}
```

1-27



## Behind the Scenes

- What you write is what you get
- Implicitly parallel
  - threads into warps
  - warps into thread groups
  - thread groups into a grid
- Hardware thread scheduler
- Highly multithreaded

1-28



## Better Device-side CUDA C (1)

```
extern "C" __global__ void
jacobikernel( float* a, float* anew, float* lchange, int n, int m )
{
    int ti = threadIdx.x, tj = threadIdx.y; /* local indices */
    int i = blockIdx.x*16+ti, j = blockIdx.y*16+tj; /* global */
    __shared__ float mychange[16*16], b[18][18];
    float mya, oldmya;
    b[tj][ti] = a[(j-1)*m+i-1];
    if(ti<2) b[tj][ti+16] = a[(j-1)*m+i+15];
    if(tj<2) b[tj+16][ti] = a[(j+15)*m+i-1];
    if(ti<2&& tj<2) b[tj+16][ti+16] = a[(j+15)*m+i+15];
    oldmya = b[tj+1][ti+1];
    __syncthreads();

    mya = w0 * oldmya +
        w1 * (b[tj+1][ti] + b[tj][ti+1] +
            b[tj+1][ti+2] + b[tj+2][ti+1]) +
        w2 * (b[tj][ti] + b[tj+2][ti] +
            b[tj][ti+2] + b[tj+2][ti+2]);
    newa[j][i] = mya;
    /* this thread's "change" */
    mychange[ti+16*tj] = fabs(mya-oldmya);
    __syncthreads();
}
```

1-29

The Portland Group®

Time for a Live Demo

1-30

The Portland Group®

## Copyright Notice

© Contents copyright 2009-2011, The Portland Group, Inc. This material may not be reproduced in any manner without the expressed written permission of The Portland Group.

1-31

The Portland Group®