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

Michael Wolfe
Michael.Wolfe@pgroup.com
http://www.pgroup.com

March 2011

Part 1: Introduction

Host Architecture Features

- Register files (integer, float)
- Functional units (integer, float, address), Icache, 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)

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
Making It Faster

- Processor:
  - Faster clocks
  - More work per clock:
    - superscalar
    - VLIW
    - more cores
    - vector / SIMD instructions

- Memory
  - Latency reduction
  - Latency tolerance

AMD “Magny-Cours”

Abstracted x64+Tesla Architecture

Abstracted x64+Fermi Architecture
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

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

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

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
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)

pthreads main routine

```c
void jacobi(int threadnum)
{
    int i, j, n = 1000, m = 1000;
    double a[1000][1000], w0 = 0.1, w1 = 0.1, w2 = 0.1;
    double lchange = 0;

    for (j = 2; j < n; j++)
        for (i = 2; i < m; i++)
            lchange = max(lchange, abs(a[i][j] - newa[i][j]));

    call pthread_mutex_lock(lck);
    change = max(change, lchange);
    call pthread_mutex_unlock(lck);
}
```

pthreads subroutine

```c
void jacobi(int threadnum)
{
    int i, j, n = 1000, m = 1000;
    double a[1000][1000], w0 = 0.1, w1 = 0.1, w2 = 0.1;
    double lchange = 0;

    for (j = 2; j < n; j++)
        for (i = 2; i < m; i++)
            lchange = max(lchange, abs(a[i][j] - newa[i][j]));

    call pthread_mutex_lock(lck);
    change = max(change, lchange);
    call pthread_mutex_unlock(lck);
}
```

Jacobi Relaxation with OpenMP directives

```c
!$omp parallel private(i,j)
!$omp do reduction(max:change)
    do j = 2, n-1
        do i = 2, m-1
            lchange = 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])
                        lchange = max(lchange, abs(newa[i,j] - a[i,j]))
                    enddo
                enddo
            !$omp end parallel
            !$omp do reduction(max:change)
            change = max(change, lchange);
        enddo
    enddo
!$omp end parallel
```
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

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

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)

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
Appropriate GPU programs

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

Jacobi Relaxation

```c
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]));
    }
}
```

Host-side CUDA C GPU Control Code

```c
__device__ float change;

...

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

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();
}

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];

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
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][ti+1];

    __syncthreads();
    mya = w0 * oldmya +
        w1 * (b[tj+1][ti] + b[tj][ti+1] +
            b[tj+1][ti+1] + b[tj+2][ti+1]) +
        w2 * (b[tj][ti] + b[tj+2][ti] +
            b[tj][ti+2] + b[tj+2][ti+2]);

    news[j][i] = mya; /* this thread's "change" */

    mychange[ti+i*16] = fabs(mya-oldmya);
    __syncthreads();
}

Time for a Live Demo