Data Parallel Architecture
GPU, Cuda Programming and Compiler Transformation

Dr. A. Sahu
CSE, IIT Guwahati

GT 8 Series GPU
- GeForce-8 Series is a massively parallel computing platform
  - 12,288 concurrent threads, hardware managed
  - 128 Thread Processor cores at 1.35 GHz -> 518 GFLOPS peak
- GPU Computing features enable C on Graphics Processing Unit

GeForce 8800 GTX Technical Specs.
- 367 GFLOPS peak performance
  - 25-50 times of current high-end microprocessors
- 265 GFLOPS sustained for appropriate applications
- Massively parallel, 128 cores
  - Partitioned into 16 Multiprocessors
- Massively threaded, sustains 1000s of threads per application
- 768 MB device memory
- 1.4 GHz clock frequency
  - CPU at 3.6 GHz
- 86.4 GB/sec memory bandwidth
  - CPU at 8 GB/sec front side bus
- Multi-GPU servers available
  - SLI Quad high-end NVIDIA GPUs on a single motherboard
  - Also 8-GPU servers announced recently

CUDA architecture
- Grid of thread blocks
- Multiple thread blocks, many warps of threads
- 240 shader cores
- 1.48 transistors
- Up to 2GB onboard memory
- ~150GB/sec BW
- 1.06 TFLOPS
- CUDA and OpenCL support
- Programmable memory spaces
- Tesla S1070 provides 4 GPUs in a 1U unit

CUDA Core
- SP = Stream Processor
- TF = Texture Filter
- IU = Instruction Unit
- SFU = Special Fun Unit

Stream multiprocessor
- Shared Memory
- Shared Memory
Graphical Processing Units

- Given the hardware invested to do graphics well, how can we supplement it to improve performance of a wider range of applications?

  - Basic idea:
    - Heterogeneous execution model
      - CPU is the host, GPU is the device
    - Develop a C-like programming language for GPU
    - Unify all forms of GPU parallelism as CUDA threads
    - Prog. model is "Single Instruction Multiple Thread"

NVIDIA GPU Architecture

- Similarities to vector machines:
  - Works well with data-level parallel problems
  - Scatter-gather transfers
  - Mask registers
  - Large register files

- Differences:
  - No scalar processor
  - Uses multithreading to hide memory latency
  - Has many functional units, as opposed to a few deeply pipelined units like a vector processor

DAXPY in C and Cuda

```c
// DAXPY(n,2.0,x,y);
// function in C

void DAXPY(int n, double a, double *x, double *y)
{
    for(int i=0; i<n; i++)
        y[i]=a*x[i]+y[i];
}
```

```c
__device__ void DAXPY(int n, double a, double *x, double *y)
{
    int i;
    i=blockIdx.x*blockDim.x + threadIdx.x;
    if (i<n)
        y[i]=a*x[i]+y[i];
}
```

Example: Multiply two vectors of length 8192

- Code that works over all elements is the grid
- Thread blocks break this down into manageable sizes — 512 threads per block
- SIMD instruction executes 32 elements at a time
- Thus
  - Grid Size = TotalRequiredThread/ThreadBlockSize = 8192/512=16
Example: Multiply two vectors of length 8192

- Block is analogous to a strip-mined vector loop with vector length of 32
- Block is assigned to a multithreaded SIMD processor by the thread block scheduler
- Current-generation GPUs (Fermi) have 7-15 multithreaded SIMD processors

CUDA Program

- CUDA program expresses data level parallelism (DLP) in terms of thread level parallelism (TLP).
- Hardware converts TLP into DLP at run time.

Scalar program

```c
float A[4][8];
do-all(i=0;i<4;i++){
    do-all(j=0;j<8;j++){
        A[i][j]++;
        A[i][j]++;
    }
}
```

CUDA program

```c
__device__ kernelF(A){
    float A[4][8];
    kernelF<<<(4,1),(8,1)>>>(A);
}
```

Two Levels of Thread Hierarchy

- The grid contains a 4x1 thread block.
- Each thread block contains 4x2 threads.

Vector Program

- Vector width is exposed to programmers.
- Scalar program (vector width of 8)

```
float A[4][8];
do-all(i=0;i<4;i++){
    do-all(j=0;j<8;j++){
        movups xmm0, [ &A[i][0] ]
        incps xmm0
        movups [ &A[i][0] ], xmm0
    }
}
```

Multi-dimension Thread and Block ID

- Both grid and thread block can have two dimensional index.

```
kernelF<<<(2,2),(4,2)>>>(A);
__device__ kernelF(A){
    i = blockDim.x * blockIdx.x + threadIdx.x;
    j = threadIdx.y + threadIdx.x;
    A[i][j]++;
}
```

Scheduling Thread Blocks on SM

- Example: Scheduling 4 thread blocks on 3 SMs.

Each thread block contains 4x2 threads.
kernelF<<<(2,2), (4,2)>>>(A);
__device__ kernelF(A){
    i = blockDim.x * blockIdx.y + blockIdx.x;
    j = threadDim.x * threadIdx.y + threadIdx.x;
    A[i][j]++;
}

Notes: the number of Processing Elements (PEs) is transparent to programmer.

Executed on machine with width of 4:

Executed on machine with width of 8: