GPUs Rock!

Nvidia Tesla K40 (Kepler)
2880 cores
4.29 Tflops (comparable to the fastest supercomputer in 2000)
Price: $4400
High-Level View of a GPU
Sample GPU SIMT Code (Simplified)

CPU code

for (ii = 0; ii < 100; ++ii) {
}

CUDA code

// there are 100 threads
__global__ void KernelFunction(...) {
    int tid = blockDim.x * blockIdx.x + threadIdx.x;
    int varA = aa[tid];
    int varB = bb[tid];
    C[tid] = varA + varB;
}
Sample GPU Program (Less Simplified)

**CPU Program**

```c
void add_matrix
( float *a, float* b, float *c, int N) {
    int index;
    for (int i = 0; i < N; ++i)
        for (int j = 0; j < N; ++j) {
            index = i + j*N;
            c[index] = a[index] + b[index];
        }
}

int main () {
    add_matrix (a, b, c, N);
}
```

**GPU Program**

```c
__global__ add_matrix
( float *a, float* b, float *c, int N) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    int j = blockIdx.y * blockDim.y + threadIdx.y;
    int index = i + j*N;
    if (i < N && j < N)
        c[index] = a[index]+b[index];
}

int main() {
    dim3 dimBlock( blocksize, blocksize) ;
    dim3 dimGrid (N/dimBlock.x, N/dimBlock.y);
    add_matrix<<<dimGrid, dimBlock>>>( a, b, c, N);
}
```
Concept of “Thread Warps” and SIMT

- Warp: A set of threads that execute the same instruction (on different data elements) → SIMT (Nvidia-speak)
- All threads run the same code
Loop Iterations as Threads

for (i=0; i < N; i++)
C[i] = A[i] + B[i];

Scalar Sequential Code

Vectorized Code

Vector Instruction

Iter. 1
Iter. 2
SIMT Memory Access

- Same instruction in different threads uses thread id to index and access different data elements

Let’s assume N=16, warpSize=4 → 4 warps
Latency Hiding with “Thread Warps”

- **Warp:** A set of threads that execute the same instruction (on different data elements)

- **Fine-grained multithreading**
  - One instruction per thread in pipeline at a time (No branch prediction)
  - Interleave warp execution to hide latencies

- **Register values of all threads stay in register file**

- **FGMT (Fine-Grained Multi-Threading)** enables long latency tolerance
Branch Divergence Problem in Warp-based SIMD

- **SPMD Execution on SIMD Hardware**
  - NVIDIA calls this “Single Instruction, Multiple Thread” (“SIMT”) execution

Diagram showing branch divergence with threads labeled A, B, C, D, E, F, G, and a table for Thread Warp 1, 2, 3, 4.
Control Flow Problem in GPUs/SIMD

• GPU uses SIMD pipeline to save area on control logic.
  – Group scalar threads into warps

• Branch divergence occurs when threads inside warps branch to different execution paths.
Branch Divergence Handling (I)

Stack

<table>
<thead>
<tr>
<th></th>
<th>Reconv. PC</th>
<th>Next PC</th>
<th>Active Mask</th>
</tr>
</thead>
<tbody>
<tr>
<td>TOS</td>
<td>-</td>
<td>E</td>
<td>1111</td>
</tr>
<tr>
<td>TOS</td>
<td>E</td>
<td>D</td>
<td>0110</td>
</tr>
<tr>
<td>TOS</td>
<td>E</td>
<td>E</td>
<td>1001</td>
</tr>
</tbody>
</table>

Thread Warp

Common PC

Thread 1 | Thread 2 | Thread 3 | Thread 4
---------|----------|----------|----------

Time

.....
Branch Divergence Handling (II)

A;
if (some condition) {
    B;
} else {
    C;
}
D;

A; if (some condition) { B; } else { C; } D;

Control Flow Stack

<table>
<thead>
<tr>
<th>Next PC</th>
<th>Recv PC</th>
<th>Amask</th>
</tr>
</thead>
<tbody>
<tr>
<td>D</td>
<td>--</td>
<td>1111</td>
</tr>
<tr>
<td>B</td>
<td>D</td>
<td>1110</td>
</tr>
<tr>
<td>D</td>
<td>D</td>
<td>0001</td>
</tr>
</tbody>
</table>

Execution Sequence

<table>
<thead>
<tr>
<th>Time</th>
<th>A</th>
<th>C</th>
<th>B</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

One per warp

TOS
Dynamic Warp Formation/Merging

- Idea: Dynamically merge threads executing the same instruction (after branch divergence)
- Form new warp at divergence
  - Enough threads branching to each path to create full new warps
Dynamic Warp Formation/Merging

- **Idea:** Dynamically merge threads executing the same instruction (after branch divergence)

  ![Diagram](image)

Dynamic Warp Formation Example

Legend

- Execution of Warp x at Basic Block A
- Execution of Warp y at Basic Block A
- A new warp created from scalar threads of both Warp x and y executing at Basic Block D

Baseline

Dynamic Warp Formation
What About Memory Divergence?

• Modern GPUs have caches
• Ideally: Want all threads in the warp to hit (without conflicting with each other)
• Problem: One thread in a warp can stall the entire warp if it misses in the cache.

• Need techniques to
  – Tolerate memory divergence
  – Integrate solutions to branch and memory divergence
NVIDIA GeForce GTX 285

• NVIDIA-speak:
  – 240 stream processors
  – “SIMT execution”

• Generic speak:
  – 30 cores
  – 8 SIMD functional units per core
NVIDIA GeForce GTX 285 “core”

- 64 KB of storage for fragment contexts (registers)
- SIMD functional unit, control shared across 8 units
  - Yellow = multiply-add
  - Blue = multiply
- Instruction stream decode
- Execution context storage
NVIDIA GeForce GTX 285 “core”

- Groups of 32 **threads** share instruction stream (each group is a Warp)
- Up to 32 warps are simultaneously interleaved
- Up to 1024 thread contexts can be stored

64 KB of storage for thread contexts (registers)
NVIDIA GeForce GTX 285

30 cores on the GTX 285: 30,720 threads