**Data Parallel Architecture**

**GPU, CUDA Programming and Compiler Transformation**

Dr. A. Sahu

CSE, IIT Guwahati

---

**Review: How Do We Reach Here?**

NVIDIA Fermi, 512 Processing Elements (PEs)

---

**Data Parallel Architecture**

- **Single Instruction Multiple Thread (SIMT)**
  - Hide vector width using scalar threads.

---

**CUDA (Compute Unified Device Architecture)**

- **DAXPY in C and CUDA**
  - Function: DAXPY
  - Parameters: n, 2.0, x, y
  - CUDA Implementation:
    ```
    __device__ void DAXPY(int n, double *x, double *y){
        int i;
        if(i<n) y[i]=a*x[i]+y[i];
    }
    ```

---

**CUDA (Compute Unified Device Architecture)**

- **Complete DAXPY in CUDA**
  - Function: DAXPY
  - Parameters: n, 2.0, x, y
  - CUDA Implementation:
    ```
    __device__ void DAXPY(int n, double *x, double *y){
        int i;
        if(i<n) y[i]=a*x[i]+y[i];
    }
    ```
Vector width is exposed to programmers.

Scalar program

```c
float A[4][8];
for (i=0; i<4; i++){
    for (j=0; j<8; j++){
        A[i][j] = 1;
    }
}
```

CUDA program

```c
__device__ void kernelF(float A[4][8]) {
    float A[4][8];
    for (i=0; i<4; i++){
        for (j=0; j<8; j++){
            A[i][j] = 1;
        }
    }
}
```

Two Levels of Thread Hierarchy

- Grid (Global): 2D array of threads, 32x32, 192 SMs
- Thread Block: 32 threads per block, 4x8x16, 8 SMs
- Thread: Unique thread in a block

Recall Vector Arch: Multiple Lanes

- Element n of vector register A is "hardwired" to element n of vector register B
  - Allows for multiple hardware lanes
**Software: Grid, Block and Thread**

- **Grid:** A kernel
- **Block:** A grid consist of several block of threads
- **Threads:** A block consist of several thread

```c
//Kernel 1:
f for (i=0; i<1000; i++)
    A[i]++;

//Kernel 2:
f for (i=0; i<1000; i++)
    for (j=0; j<100; j++)
        X[i][j]=sqrt(X[i][j]);
```

**How to write Cuda Code Smartly**

- **Look at GPU hardware**
  - Consist of many Streaming Multiprocessor (SM)
  - ThreadBlock get scheduled on SM
  - Number of Block preferably multiple of SM
- **SM:** Streaming Multiprocessor consist many Stream processor (PE)
  - It execute a block of thread
  - SP: Streaming Processor, it execute single thread of execution
  - Thread get scheduled on SP
  - Number of Thread/Block preferably multiple of Number of SPs in a SM

**CUDA Architecture**

There are two main parts

1. **Host (CPU part)**
   - Single Program, Single Data
2. **Device (GPU part)**
   - Single Program, Multiple Data

**Terminology: Program abstraction**

- **Vectorizable Loop**
- **Body of Vectorizable Loop**
- **Sequence of SIMD operations**

<table>
<thead>
<tr>
<th>Vectorizable Loop</th>
<th>Vectorizable Loop</th>
<th>Grid</th>
<th>A vectorizable loop, executed on GPU Made up of one or more ThreadBlock that can be executed in parallel</th>
</tr>
</thead>
<tbody>
<tr>
<td>Body of Vectorizable Loop</td>
<td>Body of Vectorizable Loop</td>
<td>Thread Block</td>
<td>A vectorized loop executed on a multithreaded SIMD processor; Made up of one or more threads of SIMD Instructions, They can be communicated via Local memory</td>
</tr>
<tr>
<td>Sequence of SIMD operations</td>
<td>One iteration of Scalar loop</td>
<td>Cuda Thread</td>
<td>A vertical cut of a thread of SIMD instructions corresponding to one element executed by one SIMD lane</td>
</tr>
</tbody>
</table>

**Terminology: Machine Object**

<table>
<thead>
<tr>
<th>A Thread of SIMD Instruction</th>
<th>Thread of Vector Instruction</th>
<th>Wrap</th>
<th>A traditional thread, but it just contain just SIMD that’s are executed on a multithread SIMD processor</th>
</tr>
</thead>
<tbody>
<tr>
<td>SIMD instruction</td>
<td>Vector Instruction</td>
<td>PTX instruction (Parallel Thread Execution)</td>
<td>A single SIMD instruction executed across SIMD Lanes</td>
</tr>
</tbody>
</table>

- **GUI:** Whole hardware
  - Consist of many Streaming Multiprocessor
  - It execute a kernel
- **SM:** Streaming Multiprocessor consist many Stream processor (PE)
  - It execute a block of thread
- **SP:** Streaming Processor, it execute single thread of execution
**Terminology: Processing Hardware**

<table>
<thead>
<tr>
<th>Multithreaded SIMD Processor</th>
<th>Multithreaded Vector Processor</th>
<th>Streaming Processor</th>
<th>A multithreaded SIMD processor execute thread of SIMD instructions, independent of other SIMD processors.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thread Block Scheduler</td>
<td>Scalar Processor</td>
<td>Giga Thread Engine</td>
<td>Assign multiple ThreadBlocks to multithreaded SIMD processor</td>
</tr>
<tr>
<td>SIMD Thread Scheduler</td>
<td>Thread Scheduler in a multithreaded CPU</td>
<td>Wrap Scheduler</td>
<td>Hardware unit that schedule and issue threads of SIMD instructions when they are ready to execute/Scoreboard approach to track SIMD thread execution</td>
</tr>
<tr>
<td>SIMD Lane</td>
<td>Vector Lane</td>
<td>Thread Processor</td>
<td>A SIMD lane execute the operation s in a thread of SIMD instruction on a single element</td>
</tr>
</tbody>
</table>

**Example**

- NVIDIA GPU has 32,768 registers
  - Divided into lanes
  - Each SIMD thread is limited to 64 registers
  - SIMD thread has up to:
    - 64 vector registers of 32 32-bit elements
    - 32 vector registers of 32 64-bit elements
  - Fermi has 16 physical SIMD lanes, each containing 2048 registers

```c
if (X[i] != 0) {
    X[i] = X[i] - Y[i];
} else {
    X[i] = Z[i];
}
```

**NVIDIA GPU Memory Structures**

- Each SIMD Lane has private section of off-chip DRAM
  - “Private memory”
  - Contains stack frame, spilling registers, and private variables
- Each multithreaded SIMD processor also has local memory
  - Shared by SIMD lanes / threads within a block
- Memory shared by SIMD processors is GPU Memory
  - Host can read and write GPU memory

**Fermi Architecture Innovations**

- Each SIMD processor has
  - Two SIMD thread schedulers, two instruction dispatch units
  - 16 SIMD lanes (SIMD width=32, chime=2 cycles), 16 load-store units, 4 special function units
  - Thus, two threads of SIMD instructions are scheduled every two clock cycles
- **Imps**
  - Fast double precision, Caches for GPU memory
  - 64-bit addressing and unified address space
  - Error correcting codes
  - Faster context switching, Faster atomic instructions
Focuses on determining whether data accesses in later iterations are dependent on data values produced in earlier iterations
- Loop-carried dependence

Example 1:
```c
for (i=999; i>=0; i=i-1)
x[i] = x[i] + s;
```

Example 2:
```c
for (i=0; i<100; i=i+1) {
    A[i+1] = A[i] + C[i];   /* S1 */
    B[i+1] = B[i] + A[i+1]; /* S2 */
}
```

Example 3:
```c
for (i=0; i<100; i=i+1) {
    A[i] = A[i] + B[i];   /* S1 */
    B[i+1] = C[i] + D[i]; /* S2 */
}
```

Affine Transformation
- Affine expression: An affine map f:A \rightarrow B between two affine spaces is a map on the points that acts linearly on the vectors y = ax + b
- Affine access: Array access in loop is affine if
  - Bounds of loop are expressed as affine expressions of surrounding variables and symbolic constants
  - Index of each dimension of array is also an affine expression of surrounding variables and symbolic constants

- Example
  - Z[i], Z[i+j+1], Z[0], Z[i], Z[2*i+1,3*j-10]
  - Z[*], Z[B[i]] is not an affine access

Affine Access Example
```c
For(i=0;i<M;i++)
For(j=0;j<N;j++)
{
    X[i-1]= ...; // S1
    Y[i][j] = ...; // S2
    Y[i][j+1]= ...; // S3
    Y[1][2] = ...; // S4
    Z[1][i][2*i+j] = ...; // S5
}
```
### Dependency Test: GCD test

for(i=m;i<=n;i++)
X[P]=X[Q]+C; // Loading from index Q and //Storing at index P
//Suppose P = a*i+b and Q=c*i+d

A Loop dependence exist iff
- There are two iteration indices j and k, both are with in limit (m ≤ j ≤ n and m ≤ k ≤ n)
- The loop stores into index a*j+b and later fetch from same index c*k+d
  \[ a \cdot j + b = c \cdot k + d \]
- GCD test: \( \gcd(c,a) \) must divide \( d - b \)

### Dependency Test: Example

for(i=0;i<=100;i++)
X[2*i+3]=X[2*i]+0.5;

\[
a \cdot j + b = c \cdot k + d; \quad a=2, \quad b=3, \quad c=2, \quad d=0
\]

- GCD test: \( \gcd(c,a) \) must divide \( d - b \)
  \( \gcd(2,2)=2; \quad d-b=3 \)
- Since 2 does not divide -3 no dependence possible

### Seven Primitive Transformations
- Loop Fusion
- Loop Fission
- Re-Indexing
- Scaling
- Reversal
- Permutation
- Skewing

### Transformations: Loop Fusion

for(i=1;i<=N;i++)
Y[i]=Z[i]; //s1
for(j=1;j<=N;j++)
X[i]=Y[i]; //s2

### Transformations: Re-Indexing

for(i=1;i<=N;i++)
Y[i]=Z[i]; //s1
for(j=1;j<=N;j++)
X[i]=Y[i-1]; //s2

If \( (N>1) \)
Y[N]=Z[N];
for(p=1;p<=2*N;p++)
  {
    if(p%2==0)
      Y[p]=Z[p];  //s1
    X[p]=Y[p];  //s2
  }

for(i=1;i<=N;i++)
  {
    Y[2*i]=Z[2*i];  //S1
    X[i]=Y[i]; //S2
  }

s1: p=2*i
s2: p=j

for(j=1;j<=N;j++)
  {
    X[j]=Y[j]; //S2
  }

Scaling

S1
S2

Reversal

S1
S2

Permutation

P0
Q0
J0
I0

S1
S2

Skewing

P0
Q0
J1
I1

S1
S2

for(p=1;p<=M;p++)
  {
    for(q=1;q<=N;q++)
      {
        Z[p][q]=Z[q-1][p];
      }
  }

for(i=1;i<=N+M-1;i++)
  {
    for(j=max(1,i+M); j<=min(i,M); j++)
      {
        Z[i][j]=Z[i-1][j-1];
      }
  }

A Sahu 7