046278

GPU programming model

slides by:
Mark Silberstein
Plan

- GPU programming model
- Dot product
- Host memory
- Compilation
GPU execution model

• Thread – sequence of sequentially-executed instructions
• Each thread has *private* state: registers, *stack*
• Single Instruction Multiple Threads (SIMT) model
  – Same program, called *kernel*, is instantiated and invoked *for each thread*
  – Each thread gets unique ID
• Many threads (thousands, tens of thousands)
  – Threads wait in a hardware queue until resources become available
• All threads share access to GPU memory
Vector sum $C = A + B$

- **Sequential algorithm**
  
  For every element $i$
  
  $C[i] = A[i] + B[i]$
Vector sum $C = A + B$

- **Sequential algorithm**
  
  For every element $i$
  
  $C[i] = A[i] + B[i]$

- **Parallel algorithm**
  
  In parallel every $i$
  
  $C[i] = A[i] + B[i]$
Point-wise vector sum

```c
__global__ void sum(int* A, int* B) {
    int my = getHwThreadId();
}
```

CUDA modifier to signify GPU kernels

Get unique thread ID
Inter-thread communication

• Model versatility is determined by the ability of threads to communicate
• No communications – easy for hardware, hard for software
• Fine-grained communications – hard to make efficient in hardware, easy for software
Communications: inner product
Communications: inner product

Eventually all-to-all thread communication required

How can we make it efficient??
Compromise: Hierarchy Threads → Threadblocks

- All threads are split up into fixed-size groups: threadblocks
- Threads in a threadblock can synchronize and communicate efficiently
  - Share fast memory
  - Can use barriers
Software-Hardware mapping

Thread block

HW Scheduler

GPU

GPU memory

Core

Core

Core

Core

Thread

Thread

Thread

Thread

Thread
Execution model: stream of threadblocks

- Threadblock scheduling is non-deterministic
  - Cannot rely on a schedule for correctness

- No inter-threadblock global barrier in the programming model
  - Why?
Hierarchical Dot-Product

Threadblock 1

Threadblock 2
Slow coordination should be rare

Threadblock 1
Coarse grain task

Threadblock

Efficient communication

Fine-grain task
Dot product

```c
__global__
void vector_dotproduct_kernel(float* gA, float* gB, float* gOut) {
  __shared__ float l_res[tbsize]; // local core memory

  int tid=threadIdx.x;
  int bid=blockIdx.x;

  int offset=bid*tbsize+tid;

  l_res[tid]=gA[offset]*gB[offset];

  __syncthreads(); // wait for all products in a threadblock

  // parallel reduction
  for(int i=tbsize/2; i>0; i/=2) {
    if (tid<i) l_res[tid]=l_res[tid]+l_res[i+tid];
    __syncthreads(); // wait for all partial sums
  }
  if (tid==0) gOut[bid]=l_res[0];
}
```
Dot product

```c
__global__
void vector_dotproduct_kernel(float* gA, float* gB, float* gOut)
{
    __shared_ float l_res[tbsize]; //local core memory

    int tid=threadIdx.x;
    int bid=blockIdx.x;

    int offset=bid*tbsize+tid;

    l_res[tid]=gA[offset]*gB[offset];

    __synchthreads(); // wait for all products in a threadblock

    // parallel reduction
    for(int i=tbsize/2; i>0; i/=2)
    {
        if (tid<i) l_res[tid]=l_res[tid]+l_res[i+tid];
        __synchthreads(); // wait for all partial sums
    }

    if (tid==0) gOut[bid]=l_res[0];
}
```

Two levels of indexing

Fast sync
CPU-GPU interaction

- GPU and CPU have different memory spaces
- GPU is fully managed by a CPU
  - cannot access CPU RAM
  - cannot reallocate its own memory
  - cannot start computations
- CPU must allocate memory and transfer input \textit{before kernel invocation}
- CPU must transfer output and clean up memory \textit{after kernel termination}
GPU memory

- Device code can:
  - R/W per-thread registers
  - R/W all-shared global memory

- Host code can
  - Transfer data to/from per grid global memory

GPU Teaching Kit
CUDA Device Memory Management API functions

- cudaMalloc()
  - Allocates an object in the device global memory
  - Two parameters
    - Address of a pointer to the allocated object
    - Size of allocated object in terms of bytes

- cudaFree()
  - Frees object from device global memory
  - One parameter
    - Pointer to freed object
Host-Device Data Transfer API functions

- `cudaMemcpy()`
  - memory data transfer
  - Requires four parameters
    - Pointer to destination
    - Pointer to source
    - Number of bytes copied
    - Type/Direction of transfer
  - Transfer to device is asynchronous
Code sketch in NVIDIA CUDA

```c
__global__ void sum(int* A, int* B)
{
    int my=getHwThreadId();
}

int main(int argc, char** argv)
{
    int* d_A, * d_B;
    int A[SIZE], B[SIZE];

    cudaMalloc((void**)&d_A,SIZE_BYTES); cudaMalloc((void**)&d_B,SIZE_BYTES);

    cudaMemcpy(d_A,A,SIZE_BYTES,cudaMemcpyHostToDevice);
    cudaMemcpy(d_B,B,SIZE_BYTES,cudaMemcpyHostToDevice);

    dim3 threads_in_block(512), blocks(2);

    sum<<<blocks,threads_in_block>>>(d_A,d_B);

    cudaMemcpy(d_A,A,SIZE_BYTES,cudaMemcpyDeviceToHost);
    printf("Success");
    return 0;
}
```

CPU

GPU
Code sketch in NVIDIA CUDA

```c
__global__ void sum(int* A, int* B) {
    int my = getHwThreadId();
}

int main(int argc, char** argv) {
    int* d_A, * d_B;
    int A[SIZE], B[SIZE];

cudaMalloc((void**)&d_A, SIZE_BYTES);
cudaMalloc((void**)&d_B, SIZE_BYTES);
cudaMemcpy(d_A, A, SIZE_BYTES, cudaMemcpyHostToDevice);
cudaMemcpy(d_B, B, SIZE_BYTES, cudaMemcpyHostToDevice);
dim3 threads_in_block(512), blocks(2);
sum<<<blocks, threads_in_block>>>(d_A, d_B);
cudaDeviceSynchronize();
cudaError_t error = cudaGetLastError();
if (error != cudaSuccess) {
    fprintf(stderr, "Kernel execution failed: \%s\n", cudaGetErrorString(error));
    return 1;
}
cudaMemcpy(d_A, A, SIZE_BYTES, cudaMemcpyDeviceToHost);
printf("Success");
return 0;
}
```
Code sketch in NVIDIA CUDA

```c
__global__ void sum(int* A, int* B){
    int my=getHwThreadId();
}

int main(int argc, char** argv){
    int* d_A, * d_B;
    int A[SIZE], B[SIZE];

    cudaMalloc((void**)&d_A,SIZE_BYTES); cudaMalloc((void**)&d_B,SIZE_BYTES); 

    cudaMemcpy(d_A,A,SIZE_BYTES,cudaMemcpyHostToDevice);
    cudaMemcpy(d_B,B,SIZE_BYTES,cudaMemcpyHostToDevice);

    dim3 threads_in_block(512), blocks(2);

    sum<<<blocks,threads_in_block>>>(d_A,d_B);

    cudaMemcpy((void**)&d_A,SIZE_BYTES,cudaMemcpyDeviceToHost);
    printf("Success");
    return 0;
}
```

- Two sets of pointers
  - GPU memory allocated by CPU
- Input copied from CPU to GPU
- Pass pointers to device memory in kernel invocation
Code sketch in NVIDIA CUDA

```c
__global__ void sum(int* A, int* B){
    int my=getHwThreadId();
}

int main(int argc, char** argv){
    int* d_A, * d_B;
    int A[SIZE], B[SIZE];

    cudaMemcpy((void**)&d_A,SIZE_BYTES); cudaMalloc((void**)&d_B,SIZE_BYTES);
    cudaMemcpy(d_A,A,SIZE_BYTES,cudaMemcpyHostToDevice);
    cudaMemcpy(d_B,B,SIZE_BYTES,cudaMemcpyHostToDevice);

dim3 threads_in_block(512), blocks(2);
    sum<<<blocks,threads_in_block>>>(d_A,d_B);
    cudaMemcpy(d_A,A,SIZE_BYTES,cudaMemcpyDeviceToHost);
    printf("Success");
    return 0;
}
```

Wait until the kernel is over

Copy data back
# Code sketch in NVIDIA CUDA

```c
__global__ void sum(int* A, int* B) {
    int my = getHwThreadId();
}
```

```c
int main(int argc, char** argv) {
    int* d_A, * d_B;
    int A[SIZE], B[SIZE];

    cudaMemcpy((void**)&d_A, SIZE_BYTES); cudaMalloc((void**)&d_B, SIZE_BYTES);

    cudaMemcpy(d_A, A, SIZE_BYTES, cudaMemcpyHostToDevice);
    cudaMemcpy(d_B, B, SIZE_BYTES, cudaMemcpyHostToDevice);

    dim3 threads_in_block(512), blocks(2);

    sum<<<blocks, threads_in_block>>>(d_A, d_B);

    cudaMemcpy(d_A, A, SIZE_BYTES, cudaMemcpyDeviceToHost);
    printf("Success");
    return 0;
}
```

Check for errors!
Compiling A CUDA Program

Integrated C programs with CUDA extensions

NVCC Compiler

Host Code

Host C Compiler / Linker

Device Code (PTX)

Device Just-in-Time Compiler

Heterogeneous Computing Platform with CPUs, GPUs, etc.