Intro to accelerating general purpose programs on GPUs
Topics

• What are GPUs
• Simplified programming model
  – Example – Vector addition
• Full programming model
  – Example – Vector inner product
• Brief overview of GPU hardware
• Finale example – matrix product
GPUs are parallel processors

- Throughput-oriented hardware
- Example problem – I have 100 apples to eat
  1) “high performance”: finish one apple faster
  2) “high throughput”: finish all apples faster
- Performance = parallel hardware + scalable parallel program!
Simplified GPU model

(see an introductory paper to on my home page: «GPUs: High-performance Accelerators for Parallel Applications» )
GPU 101

CPU

Memory

GPU

Memory
GPU is a co-processor
GPU is a co-processor

CPU

GPU

Memory

Memory
Co-processor model

CPU

Computation

Memory

GPU

Kernel

Memory

GPU kernel
GPU is a co-processor
Simple GPU program

• Idea: **same set** of operations is applied to different data chunks *in parallel*

• Implementation
  - Every *thread* runs the same code on different data chunks.
  - GPU concurrently runs many parallel threads
Vector sum $A = A + B$

- Sequential algorithm
  
  For every element $i < A.size$

Vector sum $A=A+B$

- **Sequential algorithm**
  
  For every element $i < A.size$
  

- **Parallel algorithm**
  
  *In parallel for every* $i < A.size$
  
Vector sum $A=A+B$

- **Sequential algorithm**
  
  For every element $i<A.size$
  

- **Parallel algorithm**
  
  In parallel for every $i<A.size$
  

- **Execution on GPU**
  
  In parallel for every $threadId < total GPU kernel threads$
  
Vector sum $A = A + B$

- **Sequential algorithm**
  
  For every element $i < A\.size$
  

- **Parallel algorithm**
  
  In parallel for every $i < A\.size$
  

- **Execution on GPU**
  
  In parallel for every $threadId < total\.GPU\.kernel\.threads$
  
GPU programming basics
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 (tens of thousands)
  - Threads wait in a hardware queue until resources become available
- All threads share access to GPU memory
CPU control of GPU

- 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 before kernel invocation
- CPU must transfer output and clean up memory after kernel termination
Vector sum for A.size=1024

- GPU
  \[ A[\text{threadId}] = A[\text{threadId}] + B[\text{threadId}] \]

- CPU
  1. Allocate arrays in GPU memory
  2. Copy data CPU -> GPU
  3. Invoke kernel with 1024 threads
  4. Wait until complete and copy data GPU->CPU
Vector sum kernel code in CUDA

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

CUDA modifier to signify GPU kernels

Get unique thread ID

fine-grain parallelism: operation per thread
CPU code for GPU management

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

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

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

    dim3 threads_in_block(512), blocks(2);
    sum<<<blocks, threads_in_block>>>(d_A, d_B);

    cudaMemcpy(A, d_A, SIZE_BYTES, cudaMemcpyDeviceToHost);
}
```

- **CPU**
  - Pass pointers to GPU memory at kernel invocation
  - Input copied from CPU to GPU

- **GPU**
  - Two sets of pointers
  - GPU memory allocated by CPU
CPU code for GPU management

```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:
            return 1;
    }}
    cudaMemcpy(A,d_A,SIZE_BYTES,cudaMemcpyDeviceToHost);
    printf ("Success");
    return 0;
}
```
Typical code structure

```
__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);
    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;
}
```
Summary so far

- GPUs are invoked by CPUs
- GPUs have their own memory
- GPU programs are called «kernels»
- GPU threads are very fine-grained
- GPU threads are independent
Thread independence is very constrained

- Vector sum is simple – purely data parallel
- What if we need coordination between tasks?
Inter-thread communication

- Programming model usability is determined by how concurrent entities 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 same-size groups: **threadblocks**
- Threads in a threadblock can synchronize and communicate efficiently
  - Share fast memory
  - Can use barriers
Execution model: stream of threadblocks

- Threadblocks are scheduled independently
- Threads within the threadblock communicate and synchronize
- Threads across threadblocks may not have global barrier
Hierarchical Dot-Product
Slow coordination should be rare

Slow coordination

Threadblock 1

Coarse grain task

Threadblock

Efficient communication

Fine-grain task
Dot product

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];
}
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];
}
So far so good

- Split your program into independent threadblocks
- Threads in a threadblock may communicate and synchronize
So far so good

- Split your program into independent threadblocks
- Threads in a threadblock may communicate and synchronize

- Wait! we need inter-threadblock synchronization in inner product
How to finalize inner product

• Option 1: use CPU
• Option 2: use atomics
• Option 3: use multiple kernel invocations
  – kernel completion = global barrier
Dot product with atomics

```c
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;

    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) atomicAdd(gOut,l_res[0]);
}
```

All threadblocks are going to **atomically** update the global memory `gOut[0]`
GPU hardware in more details
BEGIN: Terminology break
Flynn's **HARDWARE** Taxonomy

- **S I S D**
  - Single Instruction, Single Data

- **S I M D**
  - Single Instruction, Multiple Data

- **M I S D**
  - Multiple Instruction, Single Data

- **M I M D**
  - Multiple Instruction, Multiple Data
SIMD

Lock-step execution

- Example: vector operations (SSE)
MIMD

- Example: multicores
END: Terminology break
GPU hardware characteristics

- Runs thousands of concurrent execution flows (threads) identified and explicitly programmed by a developer
GPU hardware parallelism
1. MIMD = multi-core
GPU hardware parallelism
2. SIMD = vector
3. Parallelism for latency hiding

Fine-grain multithreading

GPU memory

Execution state

Core

T1
T2
T3
Fine-grain multithreading
3. Parallelism for latency hiding

GPU memory

Core

Execution state

R 0x01

T1

T2

T3
Fine-grain multithreading
3. Parallelism for latency hiding

![Diagram showing fine-grain multithreading and GPU memory with threads T1, T2, and T3 accessing memory locations R 0x01 and R 0x04.]
Fine-grain multithreading
3. Parallelism for latency hiding
Fine-grain multithreading
3. Parallelism for latency hiding

![Diagram showing parallel execution and GPU memory access]

- GPU memory
- Core
- Execution state
- R 0x01
- R 0x04
- R 0x08
- T1
- T2
- T3
Putting it all together: 3 levels of hardware parallelism

- GPU
- GPU memory
- Core
- Core
- Core
- Core
- SIMD vector
- State 1
- State k
What is GPU thread?

[Diagram showing a GPU with cores and threads]

- GPU
- GPU memory
- Core
- Core
- Core
- Core
- SIMD vector
- Thread 1
- Thread n
- State 1
- State k
What is thread block?

- **Thread block**
  - **scratchpad memory in core**
  - **Core**
  - **Thread block**
  - **SIMD vector**
  - **GPU memory**
  - **Core**
  - **Core**
  - **Core**
  - **Core**

GPU

Intro to CUDA

Mark Silberstein
What is GPU kernel

Hardware queue

Thread block

Thread block

Thread block

GPU memory

Core

Core

Core

Core

Thread block

Thread block

Thread block

Thread block
10,000-s of concurrent threads!

NVIDIA K20x GPU: 64x14x32 = 28672 concurrent threads
Using shared memory for efficient matrix product computations (using CUDA samples)
Walk-through matrix product code

```cpp
template <int BLOCK_SIZE> __global__ void matrixMulCUDA(float *C, float *A, float *B, int wA, int wB)
{
    // Block index
    int bx = blockIdx.x;
    int by = blockIdx.y;

    // Thread index
    int tx = threadIdx.x;
    int ty = threadIdx.y;

    // Index of the first sub-matrix of A processed by the block
    int aBegin = wA * BLOCK_SIZE * by;

    // Index of the last sub-matrix of A processed by the block
    int aEnd = aBegin + wA - 1;

    // Step size used to iterate through the sub-matrices of A
    int aStep = BLOCK_SIZE;

    // Index of the first sub-matrix of B processed by the block
    int bBegin = BLOCK_SIZE * bx;

    // Step size used to iterate through the sub-matrices of B
    int bStep = BLOCK_SIZE * wB;

    // Csub is used to store the element of the block sub-matrix that is computed by the thread
    float Csub = 0;
```
Walk-through matrix product code

```c
for (int a = aBegin, b = bBegin;
    a <= aEnd;
    a += aStep, b += bStep)
{

    // Declaration of the shared memory array As used to
    // store the sub-matrix of A
    __shared__ float As[BLOCK_SIZE][BLOCK_SIZE];

    // Declaration of the shared memory array Bs used to
    // store the sub-matrix of B
    __shared__ float Bs[BLOCK_SIZE][BLOCK_SIZE];

    // Load the matrices from device memory
    // to shared memory; each thread loads
    // one element of each matrix
    As[ty][tx] = A[a + wA * ty + tx];
    Bs[ty][tx] = B[b + wB * ty + tx];

    // Synchronize to make sure the matrices are loaded
    __syncthreads();
```
Walk-through matrix product code

```c
#pragma unroll

for (int k = 0; k < BLOCK_SIZE; ++k)
{
    Csub += As[ty][k] * Bs[k][tx];
}

// Synchronize to make sure that the preceding
// computation is done before loading two new
// sub-matrices of A and B in the next iteration
__syncthreads();

// Write the block sub-matrix to device memory;
// each thread writes one element
int c = wB * BLOCK_SIZE * by + BLOCK_SIZE * bx;
C[c + wB * ty + tx] = Csub;
```
More info about CUDA

- NVIDIA programming manual
- UDACITY Interactive online course: Intro to parallel computing
  - highly recommended to skim through the first two lectures before the next class

https://www.udacity.com/course/cs344
More courses on GPUs

• 046274 – Advanced topics: GPU-accelerated systems (Mark Silberstein)
• 046006 – Advanced topics: Heterogeneous computing (Avi Mendleson)
• Projects in Accelerated Systems Lab
  – Fishbach 408 - TCE
• Graduate studies

mark@ee.technion.ac.il