High Performance Computing on GPUs

Mark Silberstein
mark@ee.technion.ac.il
Why GPUs?

(from NVIDIA)

Mark Silberstein, Technion
Is it a miracle? NO!

- Architectural solutions prefers parallelism!
- Example problem – I have 100 apples to eat
  1) “high performance”: finish one apple faster
  2) “high throughput”: finish **all apples** faster
- The 1\textsuperscript{st} option is **unsustainable**
- Performance = parallel hardware + scalable parallel program!
Simplified GPU model
GPU 101

- CPU
- Memory

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

CPU

Computation

Memory

GPU

kernel

Memory
GPU is a co-processor
Compute Matrix Product ON GPU
GPUs in ML –
Linear Algebra Accelerators

How to program a GPU to compute this?

\[ z_2 = \theta_{0,2}^{(1)} + \sum_{j=1}^{m} x_j \theta_j^{(1)}_{1,2} \]

\[ = \theta_{0,2}^{(1)} + x_1 \theta_{1,2}^{(1)} + x_2 \theta_{2,2}^{(1)} + x_m \theta_{m,2}^{(1)} \]
Simple GPU program: exploiting data parallelism

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

- Algorithmic challenge – identify data-parallel tasks

- Implementation
  - Every *thread* runs the same code on different data chunks.
  - GPU concurrently runs thousands of parallel threads
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 $i$
  
  $C[i] = A[i] + B[i]$

- **Parallel algorithm**
  
  *In parallel* For every $i$
  
  $C[i] = A[i] + B[i]$
Implementation for a vector of length 1024

- **GPU kernel** (this program runs in every thread)

\[ C[\text{threadId}] = A[\text{threadId}] + B[\text{threadId}] \]
Implementation for a vector of length 1024

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

- **CPU**
  
  1. Allocate three arrays (in GPU memory)
  2. Make data accessible to GPU (CPU->GPU copy)
  3. Invoke kernel with 1024 threads
  4. Wait until complete and make data accessible to CPU (GPU->CPU copy)
Complete example

CPU:
void vector_sum(float* A, float* B, float* C, int n)
{
    float* gA=GPU_get_reference(A);
    float* gB=GPU_get_reference(B);
    float* gC=GPU_allocate_mem(n);

    GPU_set_num_threads(n);
    // GPU will invoke n threads
    GPU_run(vector_sum_kernel(gA,gB,gC));
    GPU_retrieve(C,gC);
}

GPU:
void vector_sum_kernel(float* gA, float* gB, float*gC)
{
    int my=HardwareThreadID;
    gC[my]=gA[my]+gC[my];
}
Complete example

CPU:
void vector_sum(float* A, float* B, float* C, int n)
{
    float* gA=GPU_get_reference(A);
    float* gB=GPU_get_reference(B);
    float* gC=GPU_allocate_mem(n);

    GPU_set_num_threads(n);
    // GPU will invoke n threads
    GPU_run(vector_sum_kernel(gA,gB,gC));
    GPU_retrieve(C,gC);
}

GPU:
void vector_sum_kernel(float* gA, float* gB, float* gC)
{
    int my=HardwareThreadID;
    gC[my]=gA[my]+gC[my];
}
Complete example

CPU:
void vector_sum(float* A, float* B, float* C, int n)
{
    float* gA=GPU_get_reference(A);
    float* gB=GPU_get_reference(B);
    float* gC=GPU_allocate_mem(n);

    GPU_set_num_threads(n);
    // GPU will invoke n threads
    GPU_run(vector_sum_kernel(gA,gB,gC));
    GPU_retrieve(C,gC);
}

GPU:
void vector_sum_kernel(float* gA, float* gB, float*gC)
{
    int my=HardwareThreadID;
    gC[my]=gA[my]+gC[my];
}
BUT!

- Vector sum is simple – purely data parallel
- What if we need coordination between tasks
  Example: parallel dot product
BUT!

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

Example: parallel dot product
GPU hardware
GPU hardware parallelism
1. Multi-core
GPU hardware parallelism
2. SIMD
GPU hardware parallelism

3. Hardware multithreading

![Diagram showing GPU hardware parallelism with threads T1, T2, and T3](diagram.png)
GPU hardware parallelism
3. Hardware multithreading
GPU hardware parallelism

3. Hardware multithreading

![Diagram showing GPU hardware parallelism with multithreading concept.](image-url)
GPU hardware parallelism

3. Hardware multithreading
GPU hardware parallelism

3. Hardware multithreading
Putting it all together: 3 levels of hardware parallelism

- GPU
  - GPU memory
  - Core
    - State 1
    - State k
  - Core
  - Core
  - Core

Mark Silberstein, Technion
Takeaway 1: 100,000-s of concurrent threads!

NVIDIA Turing GPU: $64 \times 32 \times 72 = 147456$ concurrent threads
Requirements for allowing fast GPU execution

- We have enough parallelism
- We have enough space to store state per thread
- We have enough bandwidth to memory
- We have efficient scheduler to manage threads
How GPU manages threads?

• Application threads are grouped into threadblocks

• Programmer defines the number of threads / threadblocks for each run
Threadblock is a building block of GPU algorithms

- Threads inside a threadblock can communicate efficiently!
  - Share a small fast scratchpad memory
  - Can be synchronized via barriers
- Threadblocks are independent
- A program consists of many threadblocks
Dot-product: hierarchical parallelization
Decomposing into threadblocks
Takeaway: parallelism hierarchy

Threadblock 1

Coarse grain task

Threadblock

Fine-grain task
Software-Hardware mapping

GPU

GPU memory

Core

Core

Core

Core

SIMD vector

Thread block

Thread n

Thread 1
Takeaway 2: One thread is slow

~100x slower than a CPU thread
Dot product

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

    int thid=LocalThreadID;
    int tbid=ThreadBlockID;

    int offset=tbid*WG_SIZE+thid;

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

    BARRIER(); // wait for all products

    for(int i=TB_SIZE/2; i>0; i/=2)
    {
        if (thid<i) l_res[thid]=l_res[thid]+l_res[i+thid];
        BARRIER(); // wait for all partial sums
    }

    if (thid==0) gOut[tbid]=l_res[0];
}
```
Parallelism structure of GPU programs

- Having many independent tasks is not enough
- Parallel structure should map well on hardware hierarchy
Estimating application performance on high-throughput processors
Compute-bound tasks

Performance bounded by maximum ALU capacity
Memory-bound tasks

- We can fully utilize a processor only if data is available in an ALU on time
- How fast can the data be made available?
  - Assume infinite number of threads, ideal parallelization
Measure of ALU/memory ratio: Arithmetic intensity

- Number of OPs per memory(**) access
  - Vector sum: 1 operation per 3 accesses. $A=1/3$
Upper bound on performance

• For memory bound algorithms only
  (why?)

\[ \text{Perf} = \text{GPUMemBandwidth} \times A \]

\[
\text{Bytes/sec} \times \text{Ops/Bytes} = \text{Ops/sec}
\]
Vector sum performance estimate

- Example: sum of vectors, GPU GTX Titan
- $A = 1/(3 \times (4 \text{ bytes}))$, MemBW = ~70GFlop/s: Performance = ~23GFLOP/s
- For comparison: raw capacity: 4.5 TFLOPs
- Only 0.5% of computing capacity utilized!!
Integrating GPUs with applications
GPUs and data locality

Input (CPU)

Compute: Matrix Product (GPU)

Compute: SoftMax + Loss

Update weights

Compute: Matrix Product (GPU)

Forward

Backward

Mark Silberstein, Technion
GPUs and data locality

Input (CPU)

Compute: Matrix Product (GPU)

Compute: SoftMax + Loss

Update weights

Where to compute?

Forward

Backward

Mark Silberstein, Technion
Problem: separate GPU memory
Data must be on a GPU

Cost(GPU) = Cost(compute) + Cost(move data)
Data must stay on a GPU

And what if data does not fit?

Input (CPU)
Compute: Matrix Product (GPU)
Compute: SoftMax + Loss (GPU)
Update weights (GPU)
Compute: Matrix Product (GPU)
Forward
Backward

Mark Silberstein, Technion
GPUs and invocation cost

Where to compute?

Input (CPU) → Compute: Matrix Product (GPU) → SoftMax → Output (CPU)
Management overhead

- GPU invocation takes at least 30,000 single CPU core cycles
- Short GPU invocations do not pay off
  - Batch multiple invocations
Sometimes CPU is faster even when it's slower

if Cost(GPU) > Cost(CPU) => compute on CPU
GPUs are used automatically, but..

- cuBLAS, cuDNN, cuFFD
  - provide powerful GPU-accelerated functions
- TensorFlow, MxNet, Caffe,...
  - run on GPUs
- So why learn how to use GPUs?
Other interesting projects in my lab

- GPU networking
- GPU access to storage
- GPU security
- GPU interaction with Smart NICs and other accelerators
Want to hear more?

- Accelerators and accelerated systems: 236278
- Undergraduate/master projects @ Accelerated Computing Systems Lab (ACSL)

mark@ee.technion.ac.il
Fishbach 408, TCE