High Performance Computing
on GPUs

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

(from NVIDIA CUDA guide)

Mark Silberstein, Technion
GPUs make a difference

NVIDIA list of GPU-enabled applications:
http://www.nvidia.com/object/gpu-applications.html
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 is a co-processor
GPU is a co-processor

Mark Silberstein, Technion
GPU is a co-processor

CPU

Computation

Memory

GPU

tation

Memory

GPU kernel

Mark Silberstein, Technion
GPU is a co-processor
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)

\[
C[\text{threadId}] = A[\text{threadId}] + B[\text{threadId}]
\]

Per-thread hardware-supplied ID
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];
}

Mark Silberstein, Technion
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

- GPU
  - GPU memory
  - Execution state
  - T1
  - T2
  - T3
GPU hardware parallelism
3. Hardware multithreading

![Diagram of GPU hardware parallelism with multithreading and execution state.]
GPU hardware parallelism
3. Hardware multithreading

![Diagram showing GPU hardware parallelism with multithreading](image)
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
  - SIMD vector

Mark Silberstein, Technion
Software-Hardware mapping

GPU

GPU memory

Core

Core

Core

Core

Thread block

SIMD vector

Thread n

Thread 1

Mark Silberstein, Technion
Assumptions

• We have enough parallelism
  – Enough\(^1\)

• We have enough state space
  – Enough\(^2\)

• We have enough bandwidth
  – Enough\(^3\)

• We have \textit{scheduler} to manage threads
Takeaway 1: 10,000-s of concurrent threads!

NVIDIA K20x GPU: $64 \times 14 \times 32 = 28672$ concurrent threads
Takeaway 2: One thread is slow

~100x slower than a CPU thread
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
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=$\sim 70\text{GFlop/s}$: Performance= $\sim 23\text{GFLOP/s}$
• For comparison: raw capacity: 4.5 TFLOPs
• Only 0.5% of computing capacity utilized!!
Integrating GPUs with applications
1. Accelerate your hotspot!

- Amdahl's law:

  **Accelerating code fraction** $f$ **improves application performance by** $1/(1-f)$
2. Management overhead

- GPU invocation takes at least 30,000 CPU cycles
- Short GPU invocations do not pay off
  - Batch multiple invocations
3. Separate memory and working set size

CPU

Memory

100GB/s

GPU

Memory

350GB/s

12 GB/s

x20
Discrete vs. hybrid GPUs
CPU-GPU memory consistency

- Traditional: weak (release) consistency
  - Concurrent write order is not guaranteed

```c
int g_a, g_b; // initialized to 0

GPU: w(g_a,2), w(g_b,3)
CPU: r(g_b,3), r(g_a,???)
```
CPU-GPU memory consistency

- Traditional: weak (release) consistency
  - Concurrent write order is not guaranteed

int g_a, g_b; // initialized to 0

GPU: w(g_a,2), w(g_b,3), <stop kernel>
CPU: <stop kernel>, r(g_b,3), r(g_a,2)
CPU-GPU memory consistency

- New fine-grain coherence
  - Concurrent write updates use acquire-release from C++11

```c
int g_a, g_b; // initialized to 0

GPU: w(g_a,2), w(g_b,3), <release>
CPU: <acquire> r(g_b,3), r(g_a,2)
```
GPUs and systems
Building systems with GPUs is hard. Why?
CPU Implementation

While(Unhappy()){
    Read_next_image_file()
    Decide_placement()
    Remove_outliers()
}
Offloading computations to GPU

This offloading pattern is common in commodity applications

```
While(Unhappy()){
    Read_next_image_file()
    Decide_placement()
    Remove_outliers()
}
```
Offloading computations to GPU

Co-processor programming model requires kernel termination

CPU

Application

Data transfer

GPU

Kernel start

Kernel termination
Kernel start/stop overheads

Invocation latency

Cache flush

Synchronization

Mark Silberstein, Technion
Implementation complexity

- Asynchronous invocation
- Manual data reuse management
- Double buffering

CPU
- copy to GPU
- invoke
- copy to GPU
- copy to CPU

GPU
Implementation complexity

Management overhead

Asynchronous invocation

Manual data reuse management

Double buffering

CPU

GPU

Why do we need to deal with low-level system details?
The reason is....

GPUs ≠ co-processors

They need I/O OS services
GPUfs: application view

CPU1
open("shared_file")

GPU1

GPU2
write()

GPU3
mmap()

GPUfs

Host File System

Mark Silberstein, Technion
GPUfs: application view

C
PUs

GPU1
GPU2
GPU3

open("shared file")

write()
mmap()

Host File System

POSIX (CPU)-like API

System-wide shared namespace

Persistent storage

Mark Silberstein, Technion
Accelerating collage app with GPUs

- No CPU management code
- open/read from GPU
Accelerating collage app with GPUfs

Overlapping computations and transfers

Read-ahead
Accelerating collage app with GPUsfs

Random data access

Data reuse
Accelerator OS

Accelerator applications

Accelerator OS support
   (Interprocessor I/O, file system, network APIs)

Hardware support for OS
   - Manycore processors
   - FPGA
   - DSPs
   - GPUs

Accelerated Applications

OS

CPU

Mark Silberstein, Technion
Other interesting projects in my lab

- GPUnet: networking for GPUs
- MPI on GPUs
- GPU network servers
- Direct access to SSDs from a GPU
- Integrating GPUs with data analytics and stream processing
- Secure GPU-accelerated systems
- Smart NICs
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