GPU Scheduling for Real-Time Multi-Tasking Environments

Toktam Dehghani
Adviser: Prof. M. Naghibzadeh
Introduction
Introduction: Multi-Core

- Performance and energy are major concerns for computer systems.
- Improving processor clock rate
  - power and heat problems.
- multi-core technology
  - high-performance with low-energy.
  - In data-parallel, in high performance computing (HPC) and embedded applications (autonomous vehicles and robots)
Introduction: GPUs

- Graphics Processing Unit
- Many-Core Processors
  - NVIDIA GeForce GTX 580 (500 cores)

CPU threads:
- Large amount of memory per thread
- Full-featured instruction set
- 1-16 execute simultaneously
  { Run few threads, each one very fast }

CUDA threads:
- Lightweight footprint
- Full-featured instruction set
- 10,000 execute simultaneously
  { Run many threads, each one slow, => total throughput high }

Ref: http://www.nvidia.com/
Introduction: GPUs

Performance trends on the well known GPU and CPU architectures

- The GPU is about 10 times faster than the CPU
- 3-10X better computing throughputs

Introduction: GPUs

Performance trends on the well known GPU and CPU architectures

- The GPU is about 7 times more energy efficient than the CPU

Introduction: GPUs

- GPUs are applicable in many domain (for graphics and computing):
  - Cloud computing services such as Amazon EC2 (GPU clusters).
  - Embedded system: a new version of Carnegie Mellon’s autonomous vehicle equips four NVIDIA’s GPUs.
  - A case study from Stanford: the GPU can speed up computer vision applications for autonomous driving by 40 times compared to CPU execution.
  - GPU-accelerated systems achieved an order of:
    - 10x speedups for software routers
    - 20x speedups for encrypted networks
    - 15x speedups for motion planning.
# GPU in Bioinformatics

<table>
<thead>
<tr>
<th>Application</th>
<th>Description</th>
<th>Expected Speed Up</th>
<th>Multi-GPU Support</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abalone</td>
<td>Models molecular dynamics of biopolymers for simulations of proteins, DNA and ligands</td>
<td>4-29x</td>
<td>No</td>
</tr>
<tr>
<td>ACEMD</td>
<td>GPU simulation of molecular mechanics force fields, implicit and explicit solvent version only</td>
<td>160 ns/day GPU version only</td>
<td>Yes</td>
</tr>
<tr>
<td>AMBER</td>
<td>Suite of programs to simulate molecular dynamics on biomolecule</td>
<td>89.44 ns/day JAC NVE</td>
<td>Yes</td>
</tr>
<tr>
<td>BarraCUDA</td>
<td>Sequence mapping software</td>
<td>6-10x</td>
<td>Yes</td>
</tr>
<tr>
<td>CUDASW++</td>
<td>Open source software for Smith-Waterman protein database searches on GPUs</td>
<td>10-50x</td>
<td>Yes</td>
</tr>
<tr>
<td>CUDA-BLASTP</td>
<td>Accelerates NCBI BLAST for scanning protein sequence databases</td>
<td>10</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Ref: [Oxford Protein Informatics Group](http://www.blopig.com/blog/2013/04/gpgpus-for-bioinformatics/), [www.toktamdehghani.com](http://www.toktamdehghani.com)
<table>
<thead>
<tr>
<th>Software</th>
<th>Description</th>
<th>Speed</th>
<th>GPU Support</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUSHAW</td>
<td>Parallelized short read aligner</td>
<td>10x</td>
<td>Yes</td>
</tr>
<tr>
<td>DL-POLY</td>
<td>Simulate macromolecules, polymers, ionic systems, etc on a distributed memory parallel computer</td>
<td>4x</td>
<td>Yes</td>
</tr>
<tr>
<td>GPU-BLAST</td>
<td>Local search with fast k-tuple heuristic</td>
<td>3-4x</td>
<td>No</td>
</tr>
<tr>
<td>GROMACS</td>
<td>Simulation of biochemical molecules with complicated bond interactions</td>
<td>165 ns/Day DHFR</td>
<td>No</td>
</tr>
<tr>
<td>GPU-HMMER</td>
<td>Parallelized local and global search with profile Hidden Markov models</td>
<td>60-100x</td>
<td>Yes</td>
</tr>
<tr>
<td>HOOMD-Blue</td>
<td>Particle dynamics package written from the ground up for GPUs</td>
<td>2x</td>
<td>Yes</td>
</tr>
<tr>
<td>LAMMPS</td>
<td>Classical molecular dynamics package</td>
<td>3-18x</td>
<td>Yes</td>
</tr>
<tr>
<td>mCUDA-MEME</td>
<td>Ultrafast scalable motif discovery algorithm based on MEME</td>
<td>4-10x</td>
<td>Yes</td>
</tr>
<tr>
<td><strong>MUMmerGPU</strong></td>
<td>An open-source high-throughput parallel pairwise local sequence alignment program</td>
<td>13x</td>
<td>No</td>
</tr>
<tr>
<td>----------------</td>
<td>----------------------------------------------------------------------------------</td>
<td>-----</td>
<td>----</td>
</tr>
<tr>
<td><strong>NAMD</strong></td>
<td>Designed for high-performance simulation of large molecular systems</td>
<td>6.44 ns/days STMV</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td></td>
<td>585x 2050s</td>
<td></td>
</tr>
<tr>
<td><strong>OpenMM</strong></td>
<td>Library and application for molecular dynamics for HPC with GPUs</td>
<td>Implicit: 127-213 ns/day; Explicit: 18-55 ns/day DHFR</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>SeqNFind</strong></td>
<td>A commercial GPU Accelerated Sequence Analysis Toolset</td>
<td>400x</td>
<td>Yes</td>
</tr>
<tr>
<td><strong>TeraChem</strong></td>
<td>A general purpose quantum chemistry package</td>
<td>7-50x</td>
<td>Yes</td>
</tr>
<tr>
<td><strong>UGENE</strong></td>
<td>Opensource Smith-Waterman for SSE/CUDA, Suffix array based repeats finder and dotplot</td>
<td>6-8x</td>
<td>Yes</td>
</tr>
</tbody>
</table>
NVIDIA And Tokyo Tech projects (On-going life science projects):

- ultra-fast fragment mapping system for DNA sequencing.

- protein-protein docking prediction.

- protein tertiary structure prediction (3-D):
  - (accelerated 4GPU+4CPU version achieved a 13.6-fold speed-up than previous 1CPU version.)
Introduction: GPGPU

- **GPGPU**: general-purpose computing on GPUs
- which enables GPUs to be used easily for “compute” programs in addition to graphics programs.

- Emerge of programming language:
  - CUDA (Nvidia)
  - OpenCL
  - …

GPU architecture

Ref:
CUDA Kernels and Threads

- Parallel portions of an application are executed on the device as **kernels**
  - One **kernel** is executed at a time
  - Many threads execute each **kernel**

**Definitions**

- **Device** = GPU
- **Host** = CPU
- **Kernel** = function that runs on the device
Thread Batching

- Kernel launches a grid of thread blocks
  - Threads within a block cooperate via shared memory
  - Threads within a block can synchronize
  - Threads in different blocks cannot cooperate
- Allows programs to *transparently scale* to different GPUs
Transparent Scalability

- Hardware is free to schedule thread blocks on any processor
- A kernel scales across parallel multiprocessors
8-Series Architecture (G80)

- 128 thread processors execute kernel threads
- 16 multiprocessors, each contains
  - 8 thread processors
- Shared memory enables thread cooperation
Execution Model

Software

Thread

Thread Block

Hardware

Thread Processor

Threads are executed by thread processors

Multiprocessor

Thread blocks are executed on multiprocessors

Thread blocks do not migrate

Several concurrent thread blocks can reside on one multiprocessor - limited by multiprocessor resources (shared memory and register file)

Grid

Device

A kernel is launched as a grid of thread blocks

Only one kernel can execute on a device at one time
Managing Memory

- CPU and GPU have separate memory spaces
- Host (CPU) code manages device (GPU) memory:
  - Allocate / free
  - Copy data to and from device
  - Applies to *global* device memory (DRAM)
Physical Memory Layout

- “Local” memory resides in device DRAM
  - Use registers and shared memory to minimize local memory use
- Host can read and write global memory but not shared memory
Kernel Memory Access

- **Per-thread**
  - Registers: On-chip
  - Local Memory: Off-chip, uncached

- **Per-block**
  - Block:
    - Shared Memory:
      - On-chip, small
      - Fast

- **Per-device**
  - Global Memory:
    - Off-chip, large
    - Uncached
    - Persistent across kernel launches
    - Kernel I/O
The Memory Hierarchy

- Local memory in each SM
- The ability to use some of this local memory as a first-level (L1) cache for global memory references.
- The local memory is 64K in size, and can be split 16K/48K or 48K/16K between L1 cache and shared memory.
- Because the access latency to this memory is also completely predictable, algorithms can be written to interleave loads, calculations, and stores with maximum efficiency.
**The Memory Hierarchy**

- Fermi GPU is also equipped with an L2.
- The L2 cache covers GPU local DRAM as well as system memory.
- The L2 cache subsystem also implements a set of memory read-modify-write operations that are atomic.
Next-Gen GPU Architecture: Fermi
GPU Programming
Programming Model

- **three major steps for user programs to take to accelerate on the GPU:**

1. **Memory Allocation:**
   - User programs must be allocated memory spaces on the host and device memory that are required for computation.

2. **Data Copy:**
   - Input data must be copied from the host to the device memory before the GPU kernel starts on the GPU.
   - Output data is also copied back from the device to the host memory to return the computed result.

3. **Kernel Launch:**
   - GPU-accelerated program code must be launched from the CPU to the GPU at runtime
   - The GPU itself is not a control unit.

execution flow of GPU-accelerated matrix multiplication

\[ A[] \times B[] = C[]. \]

GPU programming

- **Host parts**
  - the host part is more like a main thread running on the CPU to control data copies and kernel launches.
  - C and C++

- **Device Parts**
  - **GPU programming frameworks:**
    - **Open Graphics Language (OpenGL)**
      - provides a set of library functions that allow user-space applications to program GPU
    - **Open Computing Language (OpenCL)**
      - a C-like programming language
      - It can parallelize programs conceptually on any device, such as GPUs and multi-core CPUs.
    - **Compute Unified Device Architecture (CUDA)**
      - a C-like programming language
      - It can parallelize programs like OpenCL, but is dedicated to the GPU.
    - **Hybrid Multicore Parallel Programming (HMPP)**
      - to parallelize programs conceptually on any device.
      - OpenMP also employs this programming style but is dedicated to multi-core CPUs.

---

Example: host

```c
// Host code
int main()
{
    // Allocate vectors in device memory
    size_t size = N * sizeof(float);
    float* d_A;
    cudaMalloc((void**)&d_A, size);
    float* d_B;
    cudaMalloc((void**)&d_B, size);
    float* d_C;
    cudaMalloc((void**)&d_C, size);

    // Copy vectors from host memory to device memory
    // h_A and h_B are input vectors stored in host memory
    cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);

    // Invoke kernel
    int threadsPerBlock = 256;
    int threadsPerGrid =
        (N + threadsPerBlock - 1) / threadsPerBlock;
    VecAdd<<<threadsPerGrid, threadsPerBlock>>>(d_A, d_B, d_C);

    // Copy result from device memory to host memory
    // h_C contains the result in host memory
    cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);

    // Free device memory
    cudaFree(d_A);
    cudaFree(d_B);
    cudaFree(d_C);
}
```
Example: host

```c
// Host code
int main()
{
    // Allocate vectors in device memory
    size_t size = N * sizeof(float);
    float* d_A;
    cudaMalloc((void**)&d_A, size);
    float* d_B;
    cudaMalloc((void**)&d_B, size);
    float* d_C;
    cudaMalloc((void**)&d_C, size);

    // Copy vectors from host memory to device memory
    // h_A and h_B are input vectors stored in host memory
    cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);

    // Invoke kernel
    int threadsPerBlock = 256;
    int threadsPerGrid =
        (N + threadsPerBlock - 1) / threadsPerBlock;
    VecAdd<<<threadsPerGrid, threadsPerBlock>>>(d_A, d_B, d_C);

    // Copy result from device memory to host memory
    // h_C contains the result in host memory
    cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);

    // Free device memory
    cudaFree(d_A);
    cudaFree(d_B);
    cudaFree(d_C);
}
```
Example: host

```c
// Host code
int main()
{
    // Allocate vectors in device memory
    size_t size = N * sizeof(float);
    float* d_A;
    cudaMalloc((void**)&d_A, size);
    float* d_B;
    cudaMalloc((void**)&d_B, size);
    float* d_C;
    cudaMalloc((void**)&d_C, size);

    // Copy vectors from host memory to device memory
    // h_A and h_B are input vectors stored in host memory
    cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);

    // Invoke kernel
    int threadsPerBlock = 256;
    int threadsPerGrid =
        (N + threadsPerBlock - 1) / threadsPerBlock;
    VecAdd<<<threadsPerGrid, threadsPerBlock>>>(d_A, d_B, d_C);

    // Copy result from device memory to host memory
    // h_C contains the result in host memory
    cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);

    // Free device memory
    cudaFree(d_A);
    cudaFree(d_B);
    cudaFree(d_C);
}
```
Example: host

```c
// Host code
int main()
{
    // Allocate vectors in device memory
    size_t size = N * sizeof(float);
    float* d_A;
    cudaMemcpy((void**)&d_A, size);
    float* d_B;
    cudaMemcpy((void**)&d_B, size);
    float* d_C;
    cudaMemcpy((void**)&d_C, size);

    // Copy vectors from host memory to device memory
    // h_A and h_B are input vectors stored in host memory
    cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);

    // Invoke kernel
    int threadsPerBlock = 256;
    int threadsPerGrid =
        (N + threadsPerBlock - 1) / threadsPerBlock;
    VecAdd<<<threadsPerGrid, threadsPerBlock>>>(d_A, d_B, d_C);

    // Copy result from device memory to host memory
    // h_C contains the result in host memory
    cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);

    // Free device memory
    cudaFree(d_A);
    cudaFree(d_B);
    cudaFree(d_C);
}
```
Example: host

```c
// Host code
int main()
{
    // Allocate vectors in device memory
    size_t size = N * sizeof(float);
    float* d_A;
    cudaMalloc((void**)&d_A, size);
    float* d_B;
    cudaMalloc((void**)&d_B, size);
    float* d_C;
    cudaMalloc((void**)&d_C, size);

    // Copy vectors from host memory to device memory
    // h_A and h_B are input vectors stored in host memory
    cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);

    // Invoke kernel
    int threadsPerBlock = 256;
    int threadsPerGrid =
        (N + threadsPerBlock - 1) / threadsPerBlock;
    VecAdd<<<threadsPerGrid, threadsPerBlock>>>(d_A, d_B, d_C);

    // Copy result from device memory to host memory
    // h_C contains the result in host memory
    cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);

    // Free device memory
    cudaFree(d_A);
    cudaFree(d_B);
    cudaFree(d_C);
}
```
Example: host

```c
// Host code
int main()
{
    // Allocate vectors in device memory
    size_t size = N * sizeof(float);
    float* d_A;
    cudaMalloc((void**)&d_A, size);
    float* d_B;
    cudaMalloc((void**)&d_B, size);
    float* d_C;
    cudaMalloc((void**)&d_C, size);

    // Copy vectors from host memory to device memory
    // h_A and h_B are input vectors stored in host memory
    cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);

    // Invoke kernel
    int threadsPerBlock = 256;
    int threadsPerGrid =
        (N + threadsPerBlock - 1) / threadsPerBlock;
    VecAdd<<<threadsPerGrid, threadsPerBlock>>>(d_A, d_B, d_C);

    // Copy result from device memory to host memory
    // h_C contains the result in host memory
    cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);

    // Free device memory
    cudaFree(d_A);
    cudaFree(d_B);
    cudaFree(d_C);
}
```
Example: device

- Simple example: add two arrays
- Not strange code: It is C with extensions.

```c
// Device code
__global__ void VecAdd(float* A, float* B, float* C)
{
    int i = threadIdx.x;
    if (i < N)
        C[i] = A[i] + B[i];
}
```

- Example from CUDA programming guide
Work flow

- Memory allocation
- Memory copy: Host -> GPU
- Kernel call
- Memory copy: GPU -> Host
- Free GPU memory

// Device code
__global__ void VecAdd(float* A, float* B, float* C)
{
    int i = threadIdx.x;
    if (i < N)
        C[i] = A[i] + B[i];
}
GPU Channels
GPU Channel Management

- The GPU channel is an interface that bridges across the CPU and the GPU contexts.
- It is directly attached to the dispatch unit inside the GPU.
- User programs must be allocated GPU channels.
- GPU channels are independent of each other and represent separate address spaces.
- Multiple channels are supported in most GPU architectures.
  - the NVIDIA’s Fermi architecture supports 128 channels.

GPU Channel Management

- Each channel uses two types of buffers:
  - **User Push Buffer**
    - into which the user-space runtime engine pushes GPU commands.
  - **GPU Command Group**: GPU commands are usually *grouped* as *non-preemptive* regions to match user-space atomicity assumptions.
  - The more GPU commands are included in a group:
    - the less communication is required between the CPU and the GPU.
    - it could cause long blocking durations (device driver cannot *directly preempt the executions of GPU contexts launched by preceding sets of GPU commands*).
  - **Kernel Push Buffer**
    - used for kernel primitives, such as host-device synchronization, GPU initialization, and GPU mode setting.

Figure 2: GPU command submission model.

GPU scheduling
GPU Scheduling

- GPU kernel programs are launched in first-in-first-out (FIFO) fashion
- Non preemptive

- GPU schedulers:
  - device-driver level:
    - TimeGraph (according to task priorities) [1]
    - microcontroller level (solve non preemptive problem)
  - API level:
    - SRM, CM [2]

---

TimeGraph: GPU Scheduling for Real-Time Multi-Tasking Environments

Shinpei Kato† ‡, Karthik Lakshmanan†, and Ragunathan (Raj) Rajkumar†, Yutaka Ishikawa‡

† Department of Electrical and Computer Engineering, Carnegie Mellon University
‡ Department of Computer Science, The University of Tokyo

Proceedings of the 2011 USENIX conference on USENIX annual technical conference Pages 2-2
USENIX Association Berkeley, CA, USA ©2011

http://www.ertl.jp/~shinpei/publications.html
TimeGraph

- TimeGraph supports two GPU scheduling policies:
  - Predictable-Response-Time (PRT)
  - High-Throughput (HT)

- Source code of Timegraph:
  [http://rtml.ece.cmu.edu/projects/timegraph/](http://rtml.ece.cmu.edu/projects/timegraph/)
GPU Processing Without Scheduling

- high-priority task
- low-priority task
- GPU command submission

CPU

GPU

response time

response time
TimeGraph

TimeGraph Architecture

User Space
- Applications
- OpenGL/CUDA Library
- User-space GPU Driver

Kernel Space
- GPU Command Queue
- GPU Command Scheduler
- GPU resource control
- GPU Reserve Manager
- GPU exec. time prediction
- GPU Command Profiler

Device Space
- GPU Command Group
- Submission Interface
- IRQ Handler
- Interrupt
- Graphics Processing Unit (GPU)

TimeGraph GPU scheduling

Policies:

Predictable-Response-Time (PRT)
Priority Support – Predictable Response Time (PRT) Policy

- When GPU is not idle, GPU commands are queued
- When GPU gets idle, GPU commands are dispatched

---

**Diagram:**

-High-priority task
-Low-priority task
-GPU driver

**CPU**

**GPU**

Prioritized correctly

Overhead
Predictable-Response-Time (PRT)

- This policy is make high-priority tasks responsive on the GPU.
- Provide predictable response times
- Incurs overhead to make a scheduling decision at every GPU commands.
- desktop-widget, browser-plugin, and video-player tasks are desired to use the PRT policy
TimeGraph GPU scheduling
Policies:

High-Throughput (HT)
Priority Support – High Throughput (HT) Policy

- When GPU is not idle, GPU commands are queued, only if priority is lower than current GPU context.
- When GPU gets idle, GPU commands are dispatched.

High-priority task  Low-priority task  GPU driver

Overhead reduced
High-Throughput (HT)

- HT is suitable for such tasks that should execute as fast as possible.
- The HT policy achieves high throughput for one task but may block others.
- 3-D game and interactive 3-D interfacing tasks can use the HT policy.
Interference on Time

No TimeGraph Support

Priority Support (PRT)
Overhead is acceptable for protecting GPU
Problem?

- The CPU scheduling algorithm should consider the presence of the GPU.
- for real-time systems, classical deadline-driven algorithms, such as Earliest Deadline First (EDF) are not effective as they are.

Figure 9. Deadline-driven CPU scheduling in the presence of the GPU.
Laxity driven algorithms may be useful:

- such as Earliest Deadline Zero Laxity and Earliest Deadline Critical Laxity
Conclusion: TimeGraph

- Supports GPU multi-tasking
- Arbitrarily arriving non-preemptive GPU execution model
- Focuses on prioritization among GPU applications.
- Asynchronous on-chip GPU and CPU.
- Event-driven model (use GPU to CPU interrupts)
- Inside device driver (no needs for major modifications for different runtime frameworks)
Future works

- Considering presence of the GPU and CPU in the scheduling algorithms.
- Provide a periodic preemptive GPU execution model.
- Consider other scheduling parameters in the GPU scheduling.
Thank You!
Earliest-Deadline-Zero Laxity (EDZL)

- EDZL scheduling is a variant of the well known preemptive Earliest-Deadline-First (EDF) scheduling algorithm.

- The difference is the zero laxity rule:
  - Jobs with zero laxity are given the highest priority.
  - Other jobs are ranked as in EDF.
  - Ties between jobs with equal priority are assumed to be broken arbitrarily.

- The priority scheduling policy is applied globally, so that if there are \( m \) processors and \( m \) or more ready jobs then \( m \) of the jobs with highest priority will be executing.

- Like EDF, EDZL is work conserving, meaning that a processor is never idle if there is a ready job that is not executing.