GPU programming:
1. Parallel programming models

Sylvain Collange
Inria Rennes – Bretagne Atlantique
sylvain.collange@inria.fr
Graphics processing unit (GPU)

- Made for computer games: mass market
  - Low unit price, amortized R&D
  - More and more programmability and flexibility
- Inexpensive, high-performance parallel processor
  - GPUs are everywhere, from cell phones to supercomputers
GPGPU today

- 2002: General-Purpose computation on GPU (GPGPU)
  - Scientists run non-graphics computations on GPUs
- 2014: the fastest supercomputers use GPUs/accelerators
  - GPU architectures now in high-performance computing

#1 Tianhe-2 (China)
16,000 × (2 Intel Xeon + 3 Intel Xeon Phi)

#2 Titan (USA)
18,648 × (1 AMD Opteron + 1 NVIDIA Tesla K20X)
GPGPU today

- 2002: General-Purpose computation on GPU (GPGPU)
  - Scientists run non-graphics computations on GPUs
- 2014: the fastest supercomputers use GPUs/accelerators
  - GPU architectures now in high-performance computing

Grifo04 (Brazil)
544 × (1 Intel Xeon + 2 NVIDIA Tesla M2050)
Goals of the course

- How does a GPU work?
- How to design parallel algorithms?
- How to implement them on a GPU?
- How to get good performance?

- Will focus on NVIDIA CUDA environment
  - Similar concepts in OpenCL

- Non-goals: the course is not about
  - Graphics programming (OpenGL, Direct3D)
  - High-level languages that target GPUs (OpenACC, OpenMP 4…)
  - Parallel programming for multi-core and clusters (OpenMP, MPI…)
  - Programming for SIMD extensions (SSE, AVX…)
Outline of the course

- Today: Introduction to parallel programming models
  - PRAM
  - BSP
  - Multi-BSP
- Introduction to GPU architecture
  - Let's pretend we are a GPU architect
  - How would we design a GPU?
- GPU programming
  - The software side
  - Programming model
- Performance optimization
  - Possible bottlenecks
  - Common optimization techniques
## Organization: tentative schedule

<table>
<thead>
<tr>
<th>Week</th>
<th>Course: Tuesday 1:00pm room 2014</th>
<th>Lab: Thursday 1:00pm room 2011</th>
</tr>
</thead>
<tbody>
<tr>
<td>06/10</td>
<td>Programming models</td>
<td>Parallel algorithms</td>
</tr>
<tr>
<td>13/10</td>
<td>GPU architecture 1</td>
<td>Know your GPU</td>
</tr>
<tr>
<td>20/10</td>
<td>GPU architecture 2</td>
<td>Matrix multiplication</td>
</tr>
<tr>
<td>27/10</td>
<td>GPU programming 1</td>
<td>Computing ln(2) the hard way</td>
</tr>
<tr>
<td>03/11</td>
<td>GPU programming 2</td>
<td></td>
</tr>
<tr>
<td>10/11</td>
<td>GPU optimization 1</td>
<td>Game of life</td>
</tr>
<tr>
<td>17/11</td>
<td>GPU optimization 2</td>
<td></td>
</tr>
<tr>
<td>24/11</td>
<td>Lecture by Fernando Pereira</td>
<td>Project</td>
</tr>
<tr>
<td>01/12</td>
<td>Exam</td>
<td></td>
</tr>
</tbody>
</table>

Lectures and lab exercises will be available online at: [http://homepages.dcc.ufmg.br/~sylvain.collange/gpucourse/](http://homepages.dcc.ufmg.br/~sylvain.collange/gpucourse/)
Today's course

- Goal: be aware of the major parallel models and their characteristics
- Incomplete: just an overview, not an actual programming model course
  - Just cover what we need for the rest of the course
- Informal: no formal definition, no proofs of theorems
  - Proofs by handwaving
Outline of the lecture

- Why programming models?
- PRAM
  - Model
  - Example
  - Brent's theorem
- BSP
  - Motivation
  - Description
  - Example
- Multi-BSP
- Algorithm example: parallel prefix
Why programming models?

- Abstract away the complexity of the machine
- Abstract away implementation differences (languages, architectures, runtime...)

- Focus on the algorithmic choices
  - First design the algorithm, then implement it

- Define the asymptotic algorithmic complexity of algorithms
  - Compare algorithms independently of the implementation / language / computer / operating system...
Abstract machine

- One unlimited memory, constant access time
- Basis for classical algorithmic complexity theory
  - Complexity = number of instructions executed as a function of problem size
Serial algorithms

- Can you cite examples of serial algorithms and their complexity?
Serial algorithms

- Can you cite examples of serial algorithms and their complexity?
  - Binary search: $O(\log n)$
  - Quick sort: $O(n \log n)$
  - Factorization of $n$-digit number: $O(e^n)$
  - Multiplication of $n$-digit numbers:
    $O(n \log n) < O(n \log n 2^{O(\log^* n)}) < O(n \log n \log \log n)$

- Still an active area of research!
- But for parallel machines, need parallel models
Outline of the lecture

- Why programming models?
- PRAM
  - Model
  - Example
  - Brent's theorem
- BSP
  - Motivation
  - Description
  - Example
- Multi-BSP
- Algorithm example: parallel prefix
Parallel model: PRAM

- 1 memory, constant access time
- N processors
- Synchronous steps
  - All processors run an instruction simultaneously
  - Usually the same instruction

PRAM example: SAXPY

- Given vectors X and Y of size n, compute vector $R \leftarrow a \times X + Y$ with n processors

Processor $P_i$:
- $x_i \leftarrow X[i]$ 
- $y_i \leftarrow Y[i]$ 
- $r_i \leftarrow a \times x_i + y_i$ 
- $R[i] \leftarrow r_i$
PRAM example: SAXPY

- Given vectors X and Y of size n, compute vector $R \leftarrow a \times X + Y$ with n processors

Processor $P_i$:
- $x_i \leftarrow X[i]$
- $y_i \leftarrow Y[i]$
- $r_i \leftarrow a \times x_i + y_i$
- $R[i] \leftarrow r_i$
PRAM example

- Given vectors X and Y of size n, compute vector $R \leftarrow a \times X + Y$ with n processors

Processor $P_i$:
- $x_i \leftarrow X[i]$
- $y_i \leftarrow Y[i]$
- $r_i \leftarrow a \times x_i + y_i$
- $R[i] \leftarrow r_i$
**PRAM example: SAXPY**

- Given vectors $X$ and $Y$ of size $n$, compute vector $R \leftarrow a \times X + Y$ with $n$ processors

  Processor $P_i$:
  
  $x_i \leftarrow X[i]$
  $y_i \leftarrow Y[i]$
  $r_i \leftarrow a \times x_i + y_i$
  $R[i] \leftarrow r_i$

- Complexity: $O(1)$ with $n$ processors

![Diagram of PRAM example]

---

19
Access conflicts: PRAM machine types

- Can multiple processors read the same memory location?
  - **Exclusive** Read: any 2 processors have to read different locations
  - **Concurrent** Read: 2 processors can access the same location

- Can multiple processors write the same location?
  - **Exclusive** Write: any 2 processors have to write different locations
  - **Concurrent** Write: Allow reduction operators: And, Or, Sum...

- Types of PRAM machines: EREW, CREW, CRCW...

- We assume CREW in this lecture
Reduction example

- Given a vector \( a \) of size \( n \), compute the sum of elements with \( n \) processors

- Sequential algorithm
  
  \[
  \text{for } i = 0 \text{ to } n-1 \\
  \quad \text{sum} \leftarrow \text{sum} + a[i]
  \]

- All operations are dependent: no parallelism
  
  - Need to use associativity to reorder the sum
Parallel reduction: dependency graph

We compute \( r = ((a_0 \oplus a_1) \oplus (a_2 \oplus a_3)) \oplus \ldots \oplus a_{n-1}) \ldots \)

Pairs

Groups of 4
Parallel reduction: PRAM algorithm

- First step

Processor $P_i$:
$a[i] = a[2*i] + a[2*i+1]$
Parallel reduction: PRAM algorithm

- Second step
  - Reduction on n/2 processors
  - Other processors stay idle

Processor Pi:
\[ n' = n/2 \]
\[ \text{if } i < n' \text{ then} \]
\[ a[i] = a[2i] + a[2i+1] \]
\[ \text{end if} \]
Parallel reduction: PRAM algorithm

- Complete loop

Processor Pi:
  while n > 1 do
    if i < n then
      \( a[i] = a[2*i] + a[2*i+1] \)
    end if
    n = n / 2;
  end while

- Complexity: \( O(\log(n)) \) with \( O(n) \) processors
A computer that grows with problem size is not very realistic

Assume \( p \) processors for \( n \) elements, \( p < n \)

Back to \( R \leftarrow a \times X + Y \) example

- Solution 1: group by blocks
  assign ranges to each processor

Processor \( P_i \):
for \( j = i \times \lfloor n/p \rfloor \) to \( (i+1) \times \lfloor n/p \rfloor - 1 \)
\( x_j \leftarrow X[j] \)
\( y_j \leftarrow Y[j] \)
\( r_j \leftarrow a \times x_j + y_j \)
\( R[j] \leftarrow r_j \)
Less than n processors

- A computer that grows with problem size is not very realistic
- Assume $p$ processors for $n$ elements, $p < n$
- Back to $R \leftarrow a \times X + Y$ example
  - Solution 1: group by blocks
    assign ranges to each processor

Processor $P_i$:
  for $j = i \times [n/p]$ to $(i+1) \times [n/p] - 1$
  $x_j \leftarrow X[j]$
  $y_j \leftarrow Y[j]$
  $r_j \leftarrow a \times x_j + y_j$
  $R[j] \leftarrow r_j$
Solution 2: slice and interleave

Processor Pi:
for j = i to n step p
  xj ← X[j]
yj ← Y[j]
rj ← a × xj + yj
R[j] ← rj
Solution 2: slice and interleave

Processor $P_i$:
\[
\text{for } j = i \text{ to } n \text{ step } p \\
x_j \leftarrow X[j] \\
y_j \leftarrow Y[j] \\
r_j \leftarrow a \times x_j + y_j \\
R[j] \leftarrow r_j
\]

- Equivalent in the PRAM model
  - But not in actual programming!
Brent's theorem

- Generalization for any PRAM algorithm of complexity \( C(n) \) with \( n \) processors
- Derive an algorithm for \( n \) elements, with \( p \) processors only
- Idea: each processor emulates \( n/p \) virtual processors
  - Each emulated step takes \( n/p \) actual steps
- Complexity: \( O(n/p \times C(n)) \)
Interpretation of Brent's theorem

- We can design algorithms for a variable (unlimited) number of processors
  - They can run on any machine with fewer processors with known complexity
  - There may exist a more efficient algorithm with \( p \) processors only
PRAM: wrap-up

- Unlimited parallelism
- Unlimited shared memory
- Only count operations: zero communication cost

Pros

- Simple model: high-level abstraction
  can model many processors or even hardware

Cons

- Simple model: far from implementation
Outline of the lecture

- Why programming models?
- PRAM
  - Model
  - Example
  - Brent's theorem
- BSP
  - Motivation
  - Description
  - Example
- Multi-BSP
- Algorithm example: parallel prefix
PRAM limitations

- PRAM model proposed in 1978
  - Inspired by SIMD machines of the time

- Assumptions
  - All processors synchronized every instruction
  - Negligible communication latency

- Useful as a theoretical model, but far from modern computers

ILLIAC-IV, an early SIMD machine
Modern architectures

- Modern supercomputers are clusters of computers
  - Global synchronization costs millions of cycles
  - Memory is distributed
- Inside each node
  - Multi-core CPUs, GPUs
  - Non-uniform memory access (NUMA) memory
- **Synchronization** cost at all levels

Mare Nostrum, a modern distributed memory machine
Bulk-Synchronous Parallel (BSP) model

- Assumes distributed memory
  - But also works with shared memory
  - Good fit for GPUs too, with a few adaptations
- Processors execute instructions independently
- Communications between processors are explicit
- Processors need to synchronize with each other

Superstep

- A program is a sequence of supersteps
- Superstep: each processor
  - Computes
  - Sends result
  - Receive data
- Barrier: wait until all processors have finished their superstep
- Next superstep: can use data received in previous step
Example: reduction in BSP

- Start from dependency graph again

\[
\begin{align*}
& a_0 \quad a_1 \quad a_2 \quad a_3 \quad a_4 \quad a_5 \quad a_6 \quad a_7 \\
& \downarrow \quad \downarrow \quad \downarrow \quad \downarrow \quad \downarrow \quad \downarrow \quad \downarrow \quad \downarrow \\
& \oplus \quad \oplus \quad \oplus \quad \oplus \\
& \downarrow \quad \downarrow \quad \downarrow \quad \downarrow \\
& \oplus \quad \oplus \\
& \downarrow \quad \downarrow \\
& \oplus \quad \oplus \\
& \downarrow \quad \downarrow \\
& r
\end{align*}
\]
Reduction: BSP

- Adding barriers

![Diagram]

- Communication
- Barrier
- Barrier
- Barrier
Reducing communication

- Data placement matters in BSP
- Optimization: keep left-side operand local

```
\begin{align*}
ap_{2i} & \leftarrow ap_{2i} + ap_{2i+1} \\
ap_{4i} & \leftarrow ap_{4i} + ap_{4i+2} \\
ap_{8i} & \leftarrow ap_{8i} + ap_{8i+4} \\
\end{align*}
```
BSP takes synchronization cost into account
Explicit communications
Homogeneous communication cost: does not depend on processors
Outline of the lecture

- Why programming models?
- PRAM
  - Model
  - Example
  - Brent's theorem
- BSP
  - Motivation
  - Description
  - Example
- Multi-BSP
- Algorithm example: parallel prefix
The world is not flat

It is hierarchical!
Multi-BSP model

- Multi-BSP: BSP generalization with groups of processors in multiple nested levels

- Higher level: more expensive synchronization
- Arbitrary number of levels

Multi-BSP and multi-core

- Minimize communication cost on hierarchical platforms
  - Make parallel program hierarchical too
  - Take thread *affinity* into account

- On clusters (MPI): add more levels **up**
- On GPUs (CUDA): add more levels **down**
Reduction: multi-BSP

- Break into 2 levels

```
\begin{align*}
  P_0 & \quad P_1 \quad P_2 \quad P_3 \\
  a_0 & \quad a_1 \quad a_2 \quad a_3 \\
  \oplus & \quad \oplus \quad \oplus \quad \oplus \\
  a_0 & \quad a_4 \\
  \oplus & \quad \oplus \\
  a_0 & \quad a_4 \\
  \oplus & \\
  r & \\
\end{align*}
```

L1 barrier

L1 barrier

L2 Barrier

Level 1 reduction

Level 2 reduction
Recap

- **PRAM**
  - Single shared memory
  - Many processors in lockstep

- **BSP**
  - Distributed memory, message passing
  - Synchronization with barriers

- **Multi-BSP**
  - BSP with multiple scales

More abstract

Closer to implementation
Outline of the lecture

- Why programming models?
- PRAM
  - Model
  - Example
  - Brent's theorem
- BSP
  - Motivation
  - Description
  - Example
- Multi-BSP
- Algorithm example: parallel prefix
Work efficiency

- **Work**: number of operations
- **Work efficiency**: ratio between work of sequential algorithm and work of parallel algorithm
- **Example**: reduction
  - Sequential sum: $n-1$ operations
  - Parallel reduction: $n-1$ operations
  - Work efficiency: 1
  - This is the best case!
Parallel prefix

- Problem: on a given vector \( a \), for each element, compute the sum \( s_i \) of all preceding elements

\[
s_i = \sum_{k=0}^{i} a_k
\]

- Result: vector \( s = (s_0, s_1, \ldots s_{n-1}) \)

- “Sum” can be any associative operator

- Applications: stream compaction, graph algorithms, addition in hardware…

Mark Harris, Shubhabrata Sengupta, and John D. Owens. "Parallel prefix sum (scan) with CUDA." GPU gems 3, 2007
Sequential prefix: graph

- **Sequential algorithm**

\[
\text{sum} \leftarrow 0 \\
\text{for } i = 0 \text{ to } n \text{ do} \\
\quad \text{s}[i] \leftarrow \text{sum} \\
\quad \text{sum} \leftarrow \text{sum} + a[k] \\
\text{end for}
\]

- **Can we parallelize it?**
  - Of course we can:
    - this is a parallel programming course!
  - Add more computations to reduce the depth of the graph
A simple parallel prefix algorithm

- Also known as Kogge-Stone in hardware

\[ \Sigma_{i-j} \text{ is the sum } \sum_{k=i}^{j} a_k \]

\[ s[i] \leftarrow a[i] \]
\[ \text{if } i \geq 1 \text{ then } s[i] \leftarrow s[i-1] + s[i] \]

\[ \text{if } i \geq 2 \text{ then } s[i] \leftarrow s[i-2] + s[i] \]

\[ \text{if } i \geq 4 \text{ then } s[i] \leftarrow s[i-4] + s[i] \]

\[ \text{Step d: if } i \geq 2^d \text{ then } s[i] \leftarrow s[i-2^d] + s[i] \]

Analysis

- Complexity
  - $\log_2(n)$ steps with $n$ processors

- Work efficiency
  - $O(n \log_2(n))$ operations
  - $1/\log_2(n)$ work efficiency

- All communications are global
  - Bad fit for Multi-BSP model
Another parallel prefix algorithm

- Two passes
- First, upsweep: parallel reduction
  - Compute partial sums
  - Keep “intermediate” (non-root) results

Second pass: downsweep

- Set root to zero
- Go down the tree
  - Left child: propagate parent
  - Right child: compute parent + left child

\[ \begin{align*}
P0 & \quad P1 & \quad P2 & \quad P3 & \quad P4 & \quad P5 & \quad P6 & \quad P7 \\
a0 & \Sigma_{0-1} & a2 & \Sigma_{0-3} & a4 & \Sigma_{4-5} & a6 & 0 \\
\end{align*} \]

\[ \text{d=2} \]
\[ \begin{align*}
a0 & \Sigma_{0-1} & a2 & 0 & a4 & \Sigma_{4-5} & a6 & \Sigma_{0-3} \\
\end{align*} \]

\[ \text{d=1} \]
\[ \begin{align*}
a0 & 0 & a2 & \Sigma_{0-1} & a4 & \Sigma_{0-3} & a6 & \Sigma_{0-5} \\
\end{align*} \]

\[ \text{d=0} \]
\[ \begin{align*}
0 & a0 & \Sigma_{0-1} & \Sigma_{0-2} & \Sigma_{0-3} & \Sigma_{0-4} & \Sigma_{0-5} & \Sigma_{0-6} \\
\end{align*} \]

\[ s[N-1] \leftarrow 0 \]
\[ \text{for } d = \log_2(n)-1 \text{ to } 0 \text{ step -1} \]
\[ \text{if } i \mod 2^{d+1} = 2^{d+1}-1 \text{ then} \]
\[ x \leftarrow s[i] \]
\[ s[i] \leftarrow s[i-2^d] + x \]
\[ s[i-2^d] \leftarrow x \]
\[ \text{end if} \]
\[ \text{end for} \]

Hopefully nobody will notice the off-by-one difference...
Analysis

- Complexity
  - $2 \log_2(n)$ steps: $O(\log(n))$

- Work efficiency
  - Performs $2n$ operations
  - Work efficiency = $1/2$: $O(1)$

- Communications are local away from the root
  - Efficient algorithm in Multi-BSP model
Another variation: Brent-Kung

- Same reduction in upsweep phase, simpler computation in downsweep phase
- Many different topologies: Kogge-Stone, Ladner-Fischer, Han-Carlson...
  - Different tradeoffs, mostly relevant for hardware implementation

If \( i \mod 2^{d+1} = 2^{d+1} - 2^d \) then
\[
s[i] \leftarrow s[i-2^d] + s[i]
\]
end if
Given a n-D grid, compute each element as a function of its neighbors

- All updates done in parallel: no dependence
- Commonly, repeat the stencil operation in a loop

Applications: image processing, fluid flow simulation…
Stencil: naïve parallel algorithm

- 1-D stencil of size n, nearest neighbors
- m successive steps

Processor i:
for t = 0 to m
    \[ s[i] = f(s[i-1], s[i], s[i+1]) \]
end for

Needs global synchronization at each stage
Ghost zone optimization

- Idea: trade more computation for less global communication
- Affect tiles to groups of processors with overlap

![Diagram showing processors and data with overlap]

Memory for group P0-P8

Memory for group P9-P17
Ghost zone optimization

- Initialize all elements including ghost zones
- Perform local stencil computations
- Global synchronization only needed periodically

![Diagram showing the process of ghost zone optimization](image)
Wrapup

- Computation is cheap
  - Focus on communication and global memory access
  - Between communication/synchronization steps, this is familiar sequential programming

- Local communication is cheap
  - Focus on global communication
Conclusion

- This time: programming models
  - Very high level view of parallel programming
- Next time: actual GPU architecture
  - Very low level view
- Then, the main part of the course will be in-between
  - Programming: mapping an algorithm to an architecture