Experiences coding non-uniform parallelism using the CUDA GPGPU architecture

Benjamin Lerner
with Trevor Jim and Yitzhak Mandelbaum

University of Washington Computer Science and Engineering
AT&T Research

NJPLS, August 28, 2008
State of the art

- Intel CPU: 8 threads, $1500
State of the art

- Intel CPU: 8 threads, $1500
- Graphics card: 15000 threads, $500
State of the art

- Intel CPU:
  8 threads, $1500
- Graphics card:
  15000 threads, $500
- Intuitive, affordable parallelism: *priceless*
State of the art

- 2003: Brook GPU (Stanford)
- 2004: Cell (Sony)
- 2005: BSGP (MSR Asia)
- 2006: Radeon, StreamSDK (AMD)
- 2007: GeForce, CUDA SDK (NVidia)
- 2008: OpenCL (Apple), Larabee (Intel)
State of the art

- **2003**: Brook GPU (Stanford)
- **2004**: Cell (Sony)
- **2005**: Radeon, StreamSDK (AMD)
- **2006**: BSGP (MSR Asia)
- **2007**: GeForce, CUDA SDK (NVidia)
- **2008**: OpenCL (Apple), Larabee (Intel)

Experiences with CUDA
CUDA overview
  Runtime
  Organization
  Limitations

Motivating example: Earley parsing
  The algorithm
  Parallelizing the algorithm
  Challenges for parallelization

Common gotchas and workarounds
  Limited memory
  Pointer regions
  Debugging
  Code reuse

Welcome Future Improvements
Typical CUDA Program

Most programs are a mix:

- Sequential setup/result processing
- Very parallel, uniform work (kernels)
- Examples: image rendering, physics simulations, large matrix operations
Typical CUDA Program

Most programs are a mix:

► Sequential setup/result processing
► Very parallel, uniform work (kernels)
► Examples: image rendering, physics simulations, large matrix operations

Goal: offload parallel work to device
Each CUDA kernel specifies a
- grid
- of blocks
- of threads

Each kernel invocation specifies the grid and block sizes
Five levels of memory available on device:
Memory organization

Five levels of memory available on device:

- thread-local registers
- ...
Five levels of memory available on device:

- Thread-local registers
- 16KB shared memory

Declared at kernel call, unreachable from host
Memory organization

Five levels of memory available on device:

- **thread-local registers**
- **16KB shared**
- **64KB constant cache**
- **1GB global**
- **512MB texture cache**

Declared at kernel call, unreachable from host

Initialized, readable from host
Parallel execution

- Blocks execute independently
- Threads grouped within blocks:
  - 32 threads = 1 warp
  - 16 threads = 1 half-warp
- SIMD-style parallelism per half-warp, when possible:
  - “Safe” memory accesses are coalesced, execute in unison
  - Memory conflicts, branched control force serialization
Synchronization primitives

Between threads in a block:
- Barrier synchronizing all threads
- Atomic operations on shared memory

Between blocks in a kernel:
- Atomic operations on global data

Between kernels on the host:
- Barrier until kernel finishes

Note: no condition variables, semaphores, etc. as primitives...encourages certain style of coding
Memory Limitations

- All memory management is manual
- Choice of memory location is crucial
- \textit{16KB? Really?}
  - Leads to manual "paging" schemes
- Manually contort code for coalesced memory accesses
  - Crucial to performance
  - Confusing to get right
Parallelism Limitations

- No nested parallelism: can’t launch a kernel within a kernel
  - useful for different granularities of parallelism
  - … can sometimes manually fuse two or more kernels
- Synchronization primitives are difficult to use
  - Thread barriers and conditional branches don’t mix
  - Atomic operations to global memory slow entire kernel
What can be accelerated by running it on a GPU?

✔ Uniform parallelism
  - Image processing, fluid simulation, molecular dynamics, ...

✘ Sequential code
General Purpose GPU Computing

What can be accelerated by running it on a GPU?

- ✓ Uniform parallelism
  - Image processing, fluid simulation, molecular dynamics, ...

- ? Non-uniform parallelism
  - Parsing

- X Sequential code
Motivating example: Earley parsing

Input:

1 + 2 + 3
Motivating example: Earley parsing

Input:

1 + 2 + 3

CALL RET

S + S

number
Motivating example: Earley parsing

Input:

1 + 2 + 3
Motivating example: Earley parsing

Input:

1 + 2 + 3

Benjamin Lerner (UW CSE, AT&T)
Motivating example: Earley parsing

Input: \begin{align*}
&1 + 2 + 3 \\
&\text{CALL} \quad \text{CALL} \quad \text{RET} \\
&\text{S} \quad + \quad \text{S} \\
&\text{number}
\end{align*}
Motivating example: Earley parsing

Input: 1 + 2 + 3

CALL RET
CALL RET
number

Benjamin Lerner (UW CSE, AT&T) Experiences with CUDA NJPLS 2008
Motivating example: Earley parsing

Input:

1 + 2 + 3

CALL RET

CALL

number

RET

CALL

S + S
Motivating example: Earley parsing

Input:

1 + 2 + 3
Motivating example: Earley parsing

Input:

\[ 1 + 2 + 3 \]
Motivating example: Earley parsing

Input:

1 + 2 + 3

[Diagram of Earley parsing algorithm]
Motivating example: Earley parsing

Input:

1 + 2 + 3

[Diagram of Earley parser states and transitions]

CALL RET

CALL number

RET
Motivating example: Earley parsing

Input:

```
1 + 2 + 3
```

[Diagram of Earley parsing algorithm]

Benjamin Lerner (UW CSE, AT&T)
Motivating example: Earley parsing

Input:

1 + 2 + 3
Motivating example: Earley parsing

Input:

1 + 2 + 3
Parallelizing Earley parsing
Parallelizing Earley parsing
Parallelizing Earley Parsing

Benjamin Lerner (UW CSE, AT&T)
Parallelizing Earley parsing
Parallelizing Earley parsing
Challenges parallelizing Earley parsing (1)

- Memory limitations:
  - Each kernel has fixed storage for the computed items and links
  - Long input will exceed allocated storage
  - Ambiguous input might exceed allocated storage
  - ... but we don't know that before starting the kernel!
Synchronization limitations:

- Each item entails an unknown number of further items
- This is non-regular parallelism
- ... which breaks coalescing of memory accesses
- ... which breaks warp parallelism
- Barriers aren’t flexible enough for this
Problem: out of room!

- Computers have a finite amount of memory
- All algorithms have a “hot set” of actively-used memory and a “cold set” of less-active but still needed memory.
- Total needed memory may be greater than available space

Solution: virtual memory and paging
Solution: virtual memory

- Allocate space in many areas of memory
- Would like a default memory manager
  - Might handle coalescing constraints automatically?
- Only manage paging manually when necessary

Limitations:

- Like garbage collection — usually useful, but can be improved upon with expert knowledge
Problem: where’s that pesky pointer?

- Device and host have separate physical memory
  - Separate address spaces
- Device and host pointers both have type $\tau^*$
  - C has only one address space
- Reading device pointers from host: segfault
- Reading host pointers from device: kernel crash
- “Workaround”: name your variables carefully and don’t get confused

Solution: need region analysis of pointers
Problem: where’s that pesky pointer?

- Pointers to pointers are useful:
  - Traversing a worklist with “start” and “end” pointers
  - Implementing ragged-edge arrays
- Compiler currently chokes on these (?!)
- “Workaround”: use **indices** instead of pointers and don’t get confused

**Solution:** need **better pointer support and bounds checking**
Solution: need compiler analyses

Limitations of above workarounds:

▶ Compiler doesn’t check for naming consistency
▶ Compiler doesn’t do bounds-checking of offsets
▶ Relies too much on convention and coding style

Pointer analyses are central to good compilers
▶ Ought to get this right!
Problem: how do I debug this?

- Kernels run on the device
- `printf` runs on the host
- Device and host memory spaces are separate
- ...so no debug `printf`!
- ...and no breakpoints inside kernels!

Solution: use **emulation mode**
Solution: emulation mode

▶ Recompile the code to use threads on host
▶ Can use printf and breakpoints

Limitations:
▶ Scheduling threads ≠ true parallelism
▶ Host and device memory spaces are merged
▶ emulation mode semantics ≠ device semantics

Need to get this right!
Problem: can I reuse your kernel?

- Suppose someone has written a great library for parallel-prefix computation
  - (Someone has; it’s called CUDPP)
- Suppose you get to a point in your kernel where you need a parallel-prefix computation
- CUDA kernels have no call stack
- “Workaround”: er, copy and paste? go back to host?

Solution: nested kernels
Solution: want nested kernels

- Would like to just call their kernel as appropriate
- No call stack means no launching kernels from within a kernel
- Inlining means you can’t even call their subroutines properly
So what are we wishing for?

Tools!

- Region-based pointer analysis
- Nested kernels
- Better static analyses of resource usage
- Bounds checking on offsets
- Better “virtual memory” support
- Automatic handling of coalescing constraints
- More, different synchronization primitives
- An accurate emulator

...