Parallel Algorithms

James Gain, Michelle Kuttel, Sebastian Wyngaard, Simon Perkins and Jason Brownbridge

{jgain | mkuttel | sperkins | jbrownbr}@cs.uct.ac.za
swyngaard@csir.co.za

3-6 May 2011
Optimized sequential algorithms often do not permit easy parallelization
- Step back and convert algorithm at a high level
- This is why automatic parallelization of sequential code fails

Standard Parallel Algorithms:
- Map
- Scatter, Gather
- Reduction
- Scan
- Search, Sort
Map

Given:
- Array or stream of data elements $A$
- Function $f(x)$

$map(A, f) = \text{applies } f(x) \text{ to all } a_i \text{ in } A$

A GPU implementation is straightforward:
- $f(x)$ is a kernel, $A$ is a 1D array, each thread operates on $a_i$

Such operations strongly supported by Functional Programming Languages
Scatter and Gather

- **Gather:**
  - Read multiple data items to a single location

- **Scatter:**
  - Write a single data item to multiple locations

- **GPU support:**
  - At a basic (slow) level using global memory
  - Faster if locality and repeated access can be exploited
Reduction

Given:
- Binary associative operator $\odot$ with identity $I$
- Ordered set $s = [a_0, a_1, ..., a_{n-1}]$ of $n$ elements

Reduce($\odot$, $s$) returns $a_0 \odot a_1 \odot ... \odot a_{n-1}$

Example: reduce(+, [3 1 7 0 4 1 6 3]) = 25

Reductions common in parallel algorithms
- Common operators are +, $\times$, min and max
Parallel Reduction

1D parallel reduction:
- Add two halves of list together repeatedly...
- ... until we’re left with a single value

\[O(\log_2 N)\text{ steps, } O(N)\text{ work}\]
Multiple Parallel 1D Reductions

- Can run many reductions in parallel
  - Use 2D Data Structure and reduce one dimension

\[ M \times N \]

\[ \text{O}(\log_2 N) \text{ steps, O}(MN) \text{ work} \]
2D Reductions

- Like 1D reductions, only reduce in both directions simultaneously

- Note: can add more than 2x2 elements per thread
Simple CUDA Reduction

- Tree-based approach used within each thread block

- Need to be able to use multiple thread blocks
  - To process very large arrays
  - To keep all multiprocessors on the GPU busy
  - Each thread block reduces a portion of the array

- But how do we communicate partial results between thread blocks?
Problem: Global Synchronization

But CUDA has no global synchronization. Why?

- Expensive to build in hardware for GPUs with high processor count
- Deadlock

Solution: decompose into multiple kernels

- Kernel launch serves as a global synchronization point
- Kernel launch has negligible HW overhead, low SW overhead
Solution: Kernel Decomposition

- Partition the data into blocks/kernels
- Code executed by each thread is identical
- Iterative kernel invocation
Scan (aka Prefix Sum)

Given:
- Binary associative operator \(\bigcirc\)
- with identity \(I\)
- Ordered set \(s = [a_0, a_1, ..., a_{n-1}]\) of \(n\) elements

\[
\text{Scan}(\bigcirc, s) \text{ returns } [I, a_0, (a_0 \bigcirc a_1), ..., (a_0 \bigcirc a_1 \bigcirc ... \bigcirc a_{n-2})]
\]

Example: \(\text{scan}(+, [3 \ 1 \ 7 \ 0 \ 4 \ 1 \ 6 \ 3]) = [0 \ 3 \ 4 \ 11 \ 11 \ 15 \ 16 \ 22]\)
Applications of Scan

- Radix sort
- Sparse matrices
- Quicksort
- Polynomial evaluation
- String comparison
- Solving recurrences
- Lexical analysis
- Tree operations
- Stream compaction
- Histograms
Naïve Parallel Scan

For \( i \) from 1 to \( \log(n) \):
- For all \( k \) in parallel where \( k \geq 2^{i-1} \)
- Element \( k \) computes
  \[ v[k] = v[k] + v[k - 2^{i-1}] \]
Naïve Parallel Scan

For $i$ from 1 to $\log(n)$:
- For all $k$ in parallel where $k \geq 2^{i-1}$
- Element $k$ computes $v[k] = v[k] + v[k-2^{i-1}]$

$i = 1$, $k \geq 1$, $2^{i-1} = 1$
Naïve Parallel Scan

For $i$ from 1 to $\log(n)$:
- For all $k$ in parallel where $k \geq 2^{i-1}$
- Element $k$ computes $v[k] = v[k] + v[k-2^{i-1}]$

$i = 2, k \geq 2, 2^{i-1} = 2$
Naïve Parallel Scan

For $i$ from 1 to $\log(n)$:
- For all $k$ in parallel where $k \geq 2^{i-1}$
- Element $k$ computes $v[k] = v[k] + v[k-2^{i-1}]$

$i = 3$, $k \geq 4$, $2^{i-1} = 4$
CUDA Scan [1]

__global__ void scan (float *g_odata, float *g_idata, int n)
{
    extern __shared__ float temp[]; // allocated on invocation

    int thid = threadIdx.x;
    int1 pout = 0, pin = 1;

    // Load input into shared memory.
    // This is exclusive scan, so shift right by one
    // and set 1st element to 0
    temp[pout*n + thid] = (thid > 0) ? g_idata[thid-1] : 0;
    __syncthreads();

    ...
for (int offset = 1; offset < n; offset *= 2) {
    pout = 1 - pout; // swap double buffer indices
    pin = 1 - pout;
    if (thid >= offset)
        temp[pout*n+thid] += temp[pin*n+thid - offset];
    else
        temp[pout*n+thid] = temp[pin*n+thid];
    __syncthreads();
}

g_odata[thid] = temp[pout*n+thid]; // write output
Less Naïve Parallel Scan

- Naïve algorithm is step-efficient, but not work-efficient
  - $O(\log n)$ steps, but $O(n \log n)$ adds
  - Sequential version is $O(n)$
  - A factor of $\log(n)$ hurts: 10x for 1024 elements!

- Dig into parallel algorithms literature for a better solution
  - See Blelloch 1990, “Prefix Sums and Their Applications”
Algorithm Design Exercise

Problem:
- You are given 20 sorted lists, each with 500,000 elements and asked to perform 50,000 searches on each list

Task:
- Design a GPU algorithm to solve this problem.
- What is the best type of memory to use? Is there any branch divergence in your solution?

Extension:
- What if the lists were unsorted?
Search

- Find a specific element $v$ in an ordered list
  - If $v$ does not exist, find next smallest element

- Hard to beat the speed of a sequential algorithm, even for large problem sizes
  - Particularly when factoring in GPU-CPU transfer

- Instead perform multiple searches in parallel
  - Each thread performs a binary search – $O(\log n)$
  - Transfer cost of the list is amortised
The constrained GPU programming environment limits viable sorting algorithms.

**Bitonic Sort**
- Form of Merge Sort [Batcher 68]
- $O((\log n)^2)$ complexity
- In practice a Bitonic sort is about 3-4 times faster (including CPU-GPU transfer) on GPU
Designing parallel algorithms

- Don’t be afraid of (seeming) wastefulness. Must introduce an $O(n)$ component, where $n$ is the number of threads.
- Conceptual Balanced Trees are a common design pattern.
- Many common sequential algorithms have parallel equivalents – reduction, scan, search, sort.