Review: Parallel Reduction (Summation)

Basic Idea

parallel iteration 1
parallel iteration 2
parallel iteration 3
Review: Parallel Reduction (Summation)

Third version -- sequential addressing

```c
for (unsigned int s=blockDim.x/2; s>0; s>>=1) {
    if (tid < s) sdata[tid] += sdata[tid + s];
    __syncthreads();
}
```

Minimized Runtime Divergence & Shared Memory Bank Conflicts
Review: Parallel Prefix Sum

- Work efficient implementation

(A). The Up-Sweep (Reduce) Phase

Complexity: $O(n)$

[Blelloch 1990]
Review: Parallel Prefix Sum

- Work efficient implementation

(B). The Down-Sweep Phase

Complexity: $O(n)$
Review: Parallel Prefix Sum

- Shared memory bank conflicts

```
int ai = offset*(2*thid+1)-1;
int bi = offset*(2*thid+2)-1;
ai += ai / NUM_BANKS;
bi += bi / NUM_BANKS;
temp[bi] += temp[ai]
```
Parallel Sort

Bitonic sort

A *bitonic sequence* is defined as a sequence with no more than one local maximum and no more than one local minimum.

*Binary split:* Divide the bitonic list into two equal halves. Compare-exchange each item on the first half with the corresponding item in the second half.

*Result:* Two bitonic sequences where the numbers in one sequence are all less than the numbers in the other sequence.
Parallel Bitonic Sort

Bitonic sort: many steps of bitonic split

Complexity: $O(n \log(n)^2)$
Parallel steps: $O(\log(n)^2)$
Parallel Bitonic Sort Code

Bitonic sort code in CUDA

```c
__global__ static void bitonicSort(int * values) {
    extern __shared__ int shared[];
    const unsigned int tid = threadIdx.x;
    shared[tid] = values[tid];
    __syncthreads();
    for (unsigned int k = 2; k <= NUM; k *= 2) {
        for (unsigned int j = k / 2; j > 0; j /= 2) {
            unsigned int ixj = tid ^ j;
            if (ixj > tid) {
                if (((tid & k) == 0) {
                    if (shared[tid] > shared[ixj])
                        swap(shared[tid], shared[ixj]);
                } else {
                    if (shared[tid] < shared[ixj])
                        swap(shared[tid], shared[ixj]);
                }
            }
            __syncthreads();
        }
    }
    values[tid] = shared[tid];
}
```
Parallel Radix Sort

Radix Sort
1. Sort for every digit from least significant to most significant bit or the other way around. No order change for the data elements in previous sorted groups during the shuffling phase.
2. Parallel prefix sum fits nicely in this framework. Every block of threads count the frequency of different digit values and their corresponding location in the same digit value group.

Designing efficient sorting algorithms for manycore GPUs (IPDPS’09)

Adopted in CUDA Thrust Library
Static Single Assignment (SSA)
Uses of SSA

- Used in almost all modern compilers
  - LLVM
  - Open 64
    - OpenUH, UPC, AMD, Loongson compiler
  - Jikes RVM
  - Java HotSpot VM
  - Mono’s Mini JIT compiler
  - Crankshaft for Chromium V8 JavaScript engine (Dec. 2010)
  - PyPy’s JIT compiler
  - Android’s Dalvik VM’s JIT compiler
  - Single-assignment C (SaC)
  - Boomerang decompiler
  - ML compiler MLton (Matthew Fluet at RIT)
  - LuaJIT
  - gimple form in GCC (version 4 in April 2005)
Static Single Assignment (SSA)

- SSA
  - each static definition defines a new name
  - each use has a single static definition
A Ø-function is a special kind of copy that selects one of its parameters.  

We assume that all Ø-functions in a block will execute at the same time: order doesn’t matter.

\[ Y_1 \leftarrow \ldots \quad Y_2 \leftarrow \ldots \]

\[ Y_3 \leftarrow \varnothing(Y_1, Y_2) \]

Real machines do not implement a Ø-function directly in hardware.
Static Single Assignment (SSA)

- SSA
  - each static definition defines a new name
  - each use has a single static definition

- Meet operation
  - \( x \leftarrow \Phi(y, z) \)
  - placement

- Naive SSA phi-insertion
  - Algorithm?
  - How many \( \Phi \) functions are needed?
SSA Construction — $\Phi$-insertion

• Naive Insertion

  1. Insert $\Phi$-functions at every join for every name appearing in the CFG
  2. Solve reaching definitions
  3. Rename each use to the definition that reaches
     (will be unique)
SSA Construction — Φ-insertion

• Naive Insertion

1. Insert $\Phi$-functions at every join for every name appearing in the CFG
2. Solve reaching definitions
3. Rename each use to the definition that reaches
   (will be unique)

• What is The Problem

1. Too many phi-functions inserted: not space or time efficient
2. How to eliminate useless phi-functions?
   Only insert phi-function at the earliest meet points of two different values!
Control Flow in Data Flow

• How to identify the earliest meeting point of two values?
• Dominance frontiers DF(n)
  • a block $f$ is in DF(n) if
    1. $n$ dominates a predecessor of $f$
    2. $n$ does not strictly dominate $f$
• the essential idea
  • $f$ is a join point
  • one of the predecessors of $f$ is dominated by $n$
SSA Algorithm

1. CFG
2. compute Dom
3. compute DF
4. insert phi
5. rename
6. reaching def
7. “destruct” SSA

\[ x_0 \leftarrow 17 - 4 \]
\[ x_1 \leftarrow a + b \]
\[ x_2 \leftarrow y - z \]
\[ x_3 \leftarrow \phi(x_2, x_0) \]
\[ x_4 \leftarrow 13 \]
\[ x_5 \leftarrow \phi(x_4, x_3) \]
\[ z \leftarrow x_5 \times q \]
\[ x_6 \leftarrow \phi(x_1, x_5) \]
\[ s \leftarrow w - x_6 \]
Dominators

• Dominator analysis
  • Dom(n) definition
    • m dominates n iff m is on every path from root node to n
  • Data flow equation
    • \( \text{DOM}(n) = \{ n \} \cup (\cap_{p \in \text{preds}(n)} \text{DOM}(p)) \)
  • Initialization
    • \( \text{DOM}(n) = \text{all}, \forall n \text{ that is not root } n_0 \)
    • \( \text{DOM}(n_0) = \{ n_0 \} \)
  • Convergence
    • Fixed point algorithm
    • Converges after a limited number of steps

\[
\begin{align*}
  x_0 &\leftarrow 17 - 4 \\
  x_1 &\leftarrow a + b \\
  x_2 &\leftarrow y - z \\
  x_3 &\leftarrow \phi(x_2, x_0) \\
  x_4 &\leftarrow 13 \\
  x_5 &\leftarrow \phi(x_4, x_3) \\
  z &\leftarrow x_5 \times q \\
  x_6 &\leftarrow \phi(x_1, x_5) \\
  s &\leftarrow w - x_6
\end{align*}
\]
Example

\[ \text{DOM}(n) = \{ n \} \cup (\bigcap_{p \in \text{preds}(n)} \text{DOM}(p)) \]

Progress of iterative solution for DOM

\[
\begin{array}{|c|c|c|c|c|c|c|c|c|}
\hline
\text{Iteration} & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\
\hline
1 & 0 & 0,1 & 0,1,2 & 0,1,3 & 0,1,3,4 & 0,1,3,5 & 0,1,3,6 & 0,1,7 \\
2 & 0 & 0,1 & 0,1,2 & 0,1,3 & 0,1,3,4 & 0,1,3,5 & 0,1,3,6 & 0,1,7 \\
\hline
\end{array}
\]

Results of iterative solution for DOM

\[
\begin{array}{|c|c|c|c|c|c|c|c|c|}
\hline
\text{DOM} & 0 & 0,1 & 0,1,2 & 0,1,3 & 0,1,3,4 & 0,1,3,5 & 0,1,3,6 & 0,1,7 \\
\hline
\text{IDOM} & - & 0 & 1 & 1 & 3 & 3 & 3 & 1 \\
\hline
\end{array}
\]
Example

Progress of iterative solution for $\text{Dom}$

<table>
<thead>
<tr>
<th>Iteration</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0,1</td>
<td>0,1,2</td>
<td>0,1,3</td>
<td>0,1,3,4</td>
<td>0,1,3,5</td>
<td>0,1,3,6</td>
<td>0,1,7</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0,1</td>
<td>0,1,2</td>
<td>0,1,3</td>
<td>0,1,3,4</td>
<td>0,1,3,5</td>
<td>0,1,3,6</td>
<td>0,1,7</td>
</tr>
</tbody>
</table>

Results of iterative solution for $\text{Dom}$

<table>
<thead>
<tr>
<th>$\text{Dom}$</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{IDom}$</td>
<td>-</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>1</td>
</tr>
</tbody>
</table>
SSA Algorithm

1. CFG
2. compute Dom
3. compute DF
4. insert phi
5. rename
6. reaching def
7. “destruct” SSA

\[ x_0 \leftarrow 17 - 4 \]
\[ x_1 \leftarrow a + b \]
\[ x_2 \leftarrow y - z \]
\[ x_3 \leftarrow \phi(x_2, x_0) \]
\[ x_4 \leftarrow 13 \]
\[ x_5 \leftarrow \phi(x_4, x_3) \]
\[ z \leftarrow x_5 \times q \]
\[ x_6 \leftarrow \phi(x_1, x_5) \]
\[ s \leftarrow w - x_6 \]
Computing Dominance Frontiers

- **Dom to DF**
  - backward alg. [EAC, Fig. 9.10]
  - forward alg., linear time [Allen & Kennedy, Fig. 4.9]

for each join \( j \)
for pred \( p \)
for all nodes \( n \) in dom tree
from \( p \) up till IDOM(\( j \))
\( j \) is in DF(\( n \))
Example

Dominance Frontiers
- $D(n)$ is fringe just beyond the region $n$ dominates
- $m \in D(n)$ : iff $n \notin (\text{Dom}(m) - \{m\})$ but $n \in \text{Dom}(p)$ for some $p \in \text{preds}(m)$.

Flow Graph

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dom</td>
<td>0</td>
<td>0,1</td>
<td>0,1,2</td>
<td>0,1,3</td>
<td>0,1,3,4</td>
<td>0,1,3,5</td>
<td>0,1,3,6</td>
<td>0,1,7</td>
</tr>
<tr>
<td>DF</td>
<td>–</td>
<td>1</td>
<td>7</td>
<td>7</td>
<td>6</td>
<td>6</td>
<td>7</td>
<td>1</td>
</tr>
</tbody>
</table>

Error in our textbook: $D(1) = \{-\}$ not 1.
Example

Computing Dominance Frontiers

- Only join points are in DF(n) for some n
- Leads to a simple, intuitive algorithm for computing dominance frontiers

For each join point x (i.e., |preds(x)| > 1)

For each CFG predecessor of x
Walk up to IDOM(x) in the dominator tree, adding x to DF(n) for each n in the walk except IDOM(x).

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>DOM</td>
<td>0</td>
<td>0,1</td>
<td>0,1,2</td>
<td>0,1,3</td>
<td>0,1,3,4</td>
<td>0,1,3,5</td>
<td>0,1,3,6</td>
<td>0,1,7</td>
</tr>
<tr>
<td>DF</td>
<td>–</td>
<td>7</td>
<td>7</td>
<td>6</td>
<td>6</td>
<td>7</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>
For a definition of $x$ defined in a block $n$, it is enough to insert $\emptyset$-functions for that definition in the blocks that are DFs of $n$. 

- If $A \in \text{Dom}(D)$, then no insertion in $D$.
- If $A \not\in \text{Dom}(D)$, then insertion in $D$ because of $B$ but not $A$.
- If $A \not\in \text{Dom}(p(D))$ and $A \in \text{Dom}(p(D))$, i.e., $D \in \text{DF}(A)$, thus, insert $\emptyset$-function in $D$. 

Diagram:

- Block $A$ dominates $x$.
- Block $B$ is a predecessor of $A$.
- Block $C$ is a successor of $A$.
- Block $D$ is a successor of $B$.
**Example**

- DF(4) is \{6\}, so \(\leftarrow\) in 4 forces \(\emptyset\)-function in 6
- \(\leftarrow\) in 6 forces \(\emptyset\)-function in DF(6) = \{7\}
- \(\leftarrow\) in 7 forces \(\emptyset\)-function in DF(7) = \{1\}
- \(\leftarrow\) in 1 forces \(\emptyset\)-function in DF(1) = \(\emptyset\) (halt)
20-Min Recess
SSA Algorithm

1. CFG
2. compute Dom
3. compute DF
4. insert phi
5. rename
6. reaching def
7. "destruct" SSA
Reaching Definitions

- A definition \( d \) of some variable \( v \) reaches operation \( i \) if and only if \( i \) uses the defined value of \( v \) before \( v \) is redefined.
- \( \text{REACHES}(n) \): the set of definitions that reach the start of node \( n \)
- \( \text{DEDEF}(n) \): the set of downward-exposed definitions in \( n \).
- \( \text{DEFKILL}(n) \): all definitions killed by a definition in \( n \).

\[
\begin{align*}
\text{REACHES}(n_0) &= \emptyset \\
\text{REACHES}(n) &= \bigcup_{m \in \text{pred}(n)} \text{DEDEF}(m) \cup (\text{REACHES}(m) \cap \text{DEFKILL}(m))
\end{align*}
\]
Renaming Process (without Reach Analysis)

Rename variables in a pre-order walk over dominator tree
( use an array of stacks, one stack per global name)

Staring with the root block, \( b \)

a.) generate unique names for each \( \emptyset \)-function
    and push them on the appropriate stacks

b.) rewrite each operation in the block
    i. Rewrite uses of global names with the current version
       (from the stack)
    ii. Rewrite definition by inventing & pushing new name

c.) fill in \( \emptyset \)-function parameters of successor blocks

d.) recurse on \( b \)'s children in the dominator tree

e.) <on exit from \( b \)> pop names generated in \( b \) from stacks

1 counter per name for subscripts

Reset the state
Renaming and Reach Analysis

Algorithm

for each global name i
  counter[i] ← 0
  stack[i] ← Ø
call Rename(n₀)

NewName(n)
  i ← counter[n]
  counter[n] ← counter[n] + 1
  push nᵢ onto stack[n]
return nᵢ

Rename(b)
  for each Ø-function in b, x ← Ø(…)
    rename x as NewName(x)
    for each operation “x ← y op z” in b
      rewrite y as top(stack[y])
      rewrite z as top(stack[z])
      rewrite x as NewName(x)
      for each successor of b in the CFG
        rewrite appropriate Ø parameters
        for each successor s of b in dom. tree
          Rename(s)
          for each operation “x ← y op z” in b
            pop(stack[x])
Example

Before processing $B_0$

Assume $a$, $b$, $c$, and $d$ defined before $B_0$

Assume $a_0$, $b_0$, $c_0$, and $d_0$ defined before $B_0$

$\text{Counters Stacks}$

$i$ has not been defined

Assume $a$, $b$, $c$, and $d$ defined before $B_0$
Example

Assume a, b, c, & d defined before B₀

Counters

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>i</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Stacks

<table>
<thead>
<tr>
<th>a₀</th>
<th>b₀</th>
<th>c₀</th>
<th>d₀</th>
<th>i₀</th>
</tr>
</thead>
</table>

End of B₀
Assume a, b, c, & d defined before B₀
Assume a, b, c, & d defined before $B_0$

Example

End of $B_2$
Example

Before starting B₃

Counters

Stacks

Assume a, b, c, & d defined before B₀
Example

End of $B_3$

Counts

Stacks

Assume $a$, $b$, $c$, & $d$ defined before $B_0$
Example

End of B₄

Assume a, b, c, & d defined before B₀
Example

End of B₅

Counters

Stacks

Assume a, b, c, & d defined before B₀
Example

Assume a, b, c, & d defined before B₀

Counters

Stacks

End of B₆
Example

Before \( B_7 \)

Assume \( a, b, c, \) & \( d \) defined before \( B_0 \)

Counters

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>i</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>4</td>
<td>6</td>
<td>6</td>
<td>2</td>
</tr>
</tbody>
</table>

Stacks

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>i</th>
</tr>
</thead>
<tbody>
<tr>
<td>a_0</td>
<td>b_0</td>
<td>c_0</td>
<td>d_0</td>
<td>i_0</td>
</tr>
<tr>
<td>a_1</td>
<td>b_1</td>
<td>c_1</td>
<td>d_1</td>
<td>i_1</td>
</tr>
<tr>
<td>a_2</td>
<td>c_2</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Example

Assume a, b, c, & d defined before B₀

Counts

Stacks

i > 100

End of B₇
After renaming

- **Semi-pruned SSA form**

Semi-pruned $\Rightarrow$ only names live in 2 or more blocks are “global names”.

Assume \( a, b, c, \) & \( d \) defined before \( B_0 \)

Pruned SSA, phi-function only inserted if the value are live, requiring reaching definition analysis
SSA Algorithm

1. CFG
2. compute Dom
3. compute DF
4. insert phi
5. rename
6. reaching def
7. “destruct” SSA

\[ x_0 \leftarrow 17 - 4 \]
\[ x_1 \leftarrow a + b \]
\[ x_2 \leftarrow y - z \]
\[ x_3 \leftarrow \phi(x_2, x_0) \]
\[ x_4 \leftarrow 13 \]
\[ x_5 \leftarrow \phi(x_4, x_3) \]
\[ z \leftarrow x_5 \times q \]
\[ x_6 \leftarrow \phi(x_1, x_5) \]
\[ s \leftarrow w - x_6 \]
SSA Deconstruction

At some point, we need executable code
- No machines implement Ø operations
- Need to fix up the flow of values

Basic idea
- Insert copies to Ø-function predecessors
- Adds lots of copies
  - Most of them coalesce away
Deconstruction Problem: Critical Edges

An edge whose source/destination has multiple successors/predecessors

\[ i_0 \leftarrow \ldots \]

\[ i_1 \leftarrow \emptyset(i_0, i_2) \]

\[ \ldots \ldots \]

\[ i_2 \leftarrow i_1 + 1 \]

\[ \ldots \]

\[ z_0 \leftarrow i_1 + \ldots \]
Deconstruction Problem: Critical Edges

An edge whose source/destination has multiple successors/predecessors
i \leftarrow 1

y \leftarrow i

i \leftarrow i + 1

z \leftarrow y + \ldots

Original Code

i_0 \leftarrow 1

i_1 \leftarrow i_0

i_2 \leftarrow i_1 + 1

i_1 \leftarrow i_2

z_0 \leftarrow i_1 + \ldots

Critical Edge Split

SSA Form, Copies Folded

i_0 \leftarrow 1

i_1 \leftarrow i_0

i_2 \leftarrow i_1 + 1

i_1 \leftarrow i_2

z_0 \leftarrow i_1 + \ldots

Copies Inserted Incorrectly

i_0 \leftarrow 1

i_1 \leftarrow i_0

i_2 \leftarrow i_1 + 1

i_1 \leftarrow i_2

z_0 \leftarrow t + \ldots

Copies Inserted Correctly
Practice Example 1

- IDOM  DF phi renaming reach deconstruct

A:  
\[
\begin{align*}
a &= 1
\end{align*}
\]

B:  
\[
\begin{align*}
a &= 2
\end{align*}
\]

C:  
\[
\begin{align*}
b &= a
\end{align*}
\]
Practice Example 2

- IDOM  DF  phi  renaming  reach  deconstruct

A: $a = 1$

B: $a = a + 1$
Practice Example 3

- IDOM    DF phi renaming reach deconstruct
Next Class

• Register Allocation & Graph Coloring