Code Optimization Techniques for Graphics Processing Units

Fernando Magno Quintão Pereira
fernando@dcc.ufmg.br
Divergence Analysis and Optimizations

• How does SIMD execution handle branches?
• How to detect divergences during program execution?
• How to predict the branches that will diverge?
• How to manually optimize code to mitigate their negative effects?
The Kingdom of Paralland

- In a galaxy far, far away, live the **Locksteppers**.
- The locksteppers are cool, easy going creatures, who live in a *monastic* and *harmonious* society.
- But, they suffer from a weird **idiosyncrasy**...
They **must** eat together...
They **must** pray together...
They **must** dance together...
For that is the law:

“All the locksteppers who do something at a given point in time, must do the same thing.”

The God of the Locksteppers
But, sometimes they disagree...

• Then, we have two parties: righties and lefties.
• According to the law, righties must sleep, while lefties do stuff.
  – When the lefties are done, they must sleep, while the righties do stuff.
  – Until they re-converge.
The Paralland Mission

Decrease the amount of time Locksteppers sleep!

• No easy task: Locksteppers disagree a lot.
• How to accomplish this mission?
• What does Paralland have to do with GPUs?
Paralland and GPU’s

• Each Lockstepper is a warp thread.
• Divergences – or disagreements – happen because of branches.

• We have a SIMD model of parallel execution.
Example of Divergent CFG

• Below we have a simple *kernel*, and its *Control Flow Graph*:

```c
__global__ void ex (float* v) {
    if (v[tid] < 0.0) {
        v[tid] /= 2;
    } else {
        v[tid] = 0.0;
    }
}
```

• Why do we have divergences in this kernel?
What does the divergence do?

<table>
<thead>
<tr>
<th>program counter</th>
<th>label</th>
<th>op</th>
<th>def</th>
<th>use₁</th>
<th>use₂</th>
<th>ALU₁</th>
<th>ALU₂</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>B₀</td>
<td>addr</td>
<td>%r₁</td>
<td>v</td>
<td>[%tid]</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>ld</td>
<td>%f₁</td>
<td>%r₁</td>
<td></td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>set.lt</td>
<td>%p₁</td>
<td>%f₁</td>
<td>0.0</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>bra</td>
<td>%p₁</td>
<td>$ZR</td>
<td></td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>5</td>
<td>BZR</td>
<td>div</td>
<td>%f₂</td>
<td>%f₁</td>
<td>2</td>
<td>●</td>
<td>✓</td>
</tr>
<tr>
<td>6</td>
<td></td>
<td>jump</td>
<td></td>
<td>$ST</td>
<td></td>
<td>●</td>
<td>✓</td>
</tr>
<tr>
<td>7</td>
<td>B₁</td>
<td>mov</td>
<td>%f₂</td>
<td>0.0</td>
<td></td>
<td>✓</td>
<td>●</td>
</tr>
<tr>
<td>8</td>
<td>Bₘₜ</td>
<td>st</td>
<td>%r₁</td>
<td>%f₂</td>
<td></td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>
To wrap up definitions...

• Why are divergences a problem in the GPU, while they are not even a concern in multi-core CPUs?
• How does latency and throughput suffer with divergences?
  – What are the bad cases in terms of performance degradation?
• Can you think about a way to detect divergences during the program execution?
The Kernels of Samuel

• What is the best input for the kernel below?

```c
__global__ void dec2zero(int* v, int N) {
    int xIndex = blockIdx.x*blockDim.x+threadIdx.x;
    if (xIndex < N) {
        while (v[xIndex] > 0) {
            v[xIndex]--;
        }
    }
}
```

From `decPar`, available in the course webpage.
Trying different inputs

void vecIncInit(int* data, int size) {
    for (int i = 0; i < size; ++i) {
        data[i] = size - i - 1;
    }
}

void vecConsInit(int* data, int size) {
    int cons = size / 2;
    for (int i = 0; i < size; ++i) {
        data[i] = cons;
    }
}

void vecAltInit(int* data, int size) {
    for (int i = 0; i < size; ++i) {
        if (i % 2) {
            data[i] = size;
        }
    }
}

void vecRandomInit(int* data, int size) {
    for (int i = 0; i < size; ++i) {
        data[i] = random() % size;
    }
}

void vecHalfInit(int* data, int size) {
    for (int i = 0; i < size/2; ++i) {
        data[i] = 0;
    }
    for (int i = size/2; i < size; ++i) {
        data[i] = size;
    }
}

• What is the best way to initialize the kernel, thus pleasing Samuel?
void vecIncInit(int* data, int size) {
    for (int i = 0; i < size; ++i) {
        data[i] = size - i - 1;
    }
}

void vecConsInit(int* data, int size) {
    int cons = size / 2;
    for (int i = 0; i < size; ++i) {
        data[i] = cons;
    }
}

void vecAltInit(int* data, int size) {
    for (int i = 0; i < size; ++i) {
        if (i % 2) {
            data[i] = size;
        }
    }
}

void vecRandomInit(int* data, int size) {
    for (int i = 0; i < size; ++i) {
        data[i] = random() % size;
    }
}

void vecHalfInit(int* data, int size) {
    for (int i = 0; i < size/2; ++i) {
        data[i] = 0;
    }
    for (int i = size/2; i < size; ++i) {
        data[i] = size;
    }
}

SUM: 20480000
TIME: 16250
Samuelic array 2

void vecIncInit(int* data, int size) {
    for (int i = 0; i < size; ++i) {
        data[i] = size - i - 1;
    }
}

void vecConsInit(int* data, int size) {
    int cons = size / 2;
    for (int i = 0; i < size; ++i) {
        data[i] = cons;
    }
}

void vecAltInit(int* data, int size) {
    for (int i = 0; i < size; ++i) {
        if (i % 2) {
            data[i] = size;
        }
    }
}

void vecRandomInit(int* data, int size) {
    for (int i = 0; i < size; ++i) {
        data[i] = random() % size;
    }
}

void vecHalfInit(int* data, int size) {
    for (int i = 0; i < size/2; ++i) {
        data[i] = 0;
    }
    for (int i = size/2; i < size; ++i) {
        data[i] = size;
    }
}

SUM: 20480000
TIME: 16250
SUM: 20480000
TIME: 16153
void vecIncInit(int* data, int size) {
    for (int i = 0; i < size; ++i) {
        data[i] = size - i - 1;
    }
}

void vecConsInit(int* data, int size) {
    int cons = size / 2;
    for (int i = 0; i < size; ++i) {
        data[i] = cons;
    }
}

void vecAltInit(int* data, int size) {
    for (int i = 0; i < size; ++i) {
        if (i % 2) {
            data[i] = size;
        }
    }
}

void vecRandomInit(int* data, int size) {
    for (int i = 0; i < size; ++i) {
        data[i] = random() % size;
    }
}

void vecHalfInit(int* data, int size) {
    for (int i = 0; i < size/2; ++i) {
        data[i] = 0;
    }
    for (int i = size/2; i < size; ++i) {
        data[i] = size;
    }
}
void vecIncInit(int *data, int size) {
    for (int i = 0; i < size; ++i) {
        data[i] = size - i - 1;
    }
}

void vecConsInit(int *data, int size) {
    int cons = size / 2;
    for (int i = 0; i < size; ++i) {
        data[i] = cons;
    }
}

void vecAltInit(int *data, int size) {
    for (int i = 0; i < size; ++i) {
        if (i % 2) {
            data[i] = size;
        }
    }
}

void vecRandomInit(int *data, int size) {
    for (int i = 0; i < size; ++i) {
        data[i] = random() % size;
    }
}

void vecHalfInit(int *data, int size) {
    for (int i = 0; i < size/2; ++i) {
        data[i] = 0;
    }
    for (int i = size/2; i < size; ++i) {
        data[i] = size;
    }
}
void vecIncInit(int* data, int size) {
    for (int i = 0; i < size; ++i) {
        data[i] = size - i - 1;
    }
}

void vecConsInit(int* data, int size) {
    int cons = size / 2;
    for (int i = 0; i < size; ++i) {
        data[i] = cons;
    }
}

void vecAltInit(int* data, int size) {
    for (int i = 0; i < size; ++i) {
        if (i % 2) {
            data[i] = size;
        }
    }
}

void vecRandomInit(int* data, int size) {
    for (int i = 0; i < size; ++i) {
        data[i] = random() % size;
    }
}

void vecHalfInit(int* data, int size) {
    for (int i = 0; i < size/2; ++i) {
        data[i] = 0;
    }
    for (int i = size/2; i < size; ++i) {
        data[i] = size;
    }
}
Profilers

• Has anyone hear of *gprof*, what about *valgrind*?
  – What do these tools measure?

• What can we do with traditional profilers?

• Profilers are mostly dynamic tools. However, there are static profilers too. Has anyone heard of anything like this?

• What is interesting to profile on GPUs?
Profiling for Divergences

Given a program, and its entry, how many times each branch has been visited, and how many divergences have happened per branch?

- How to implement a profiler that answers this question?
A Simple Solution

• We can measure divergences with a profiler that works via instrumentation

• At each divergent path, we do a referendum among all the warp threads.
  – If they all vote together, then there is no divergence.
  – If they don’t, then there is divergence.

• But we must find a writer...

Why is it difficult to find a writer?
• What is the asymptotic complexity of profiling via instrumentation?
Example: Bitonic Sort

- How to sort efficiently in parallel?
- Has anyone heard about sorting networks?
  - Shell Sort (with bubble-sort)?
  - Bitonic Sort?
__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];
}
%ixj = xor %tid %j
%p1 = gt %ixj %tid
bra %p1 L2
%t1 = and %tid %k
%p2 = eq %t1 0
bra %p2 L3
%t2 = ld %shared[%tid]
%t3 = ld %shared[%ixj]
%p3 = gt %t2 %t3
bra %p3 L7
%t4 = ld %shared[%tid]
%t5 = ld %shared[%ixj]
%p4 = lt %t4 %t5
bra %p4 L7
st %shared[%tid] %t3
st %shared[%ixj] %t2
st %shared[%tid] %t5
st %shared[%ixj] %t4
sync
The Control Flow Graph

• What are the boxes?
• What are the boldface numbers in front of the boxes?
• What are the edges?
• Can you make a correlation between source code and CFG?
• Can you think about a systematic way to produce CFGs from source code?

• Go back to the CFG and answer me: what would be the most serious divergences?
## Warp Trace

<table>
<thead>
<tr>
<th>cycle</th>
<th>label</th>
<th>opcode</th>
<th>def</th>
<th>use₁</th>
<th>use₂</th>
<th>t₀</th>
<th>t₁</th>
<th>t₂</th>
<th>t₃</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>L₁</td>
<td>xor</td>
<td>%ixj</td>
<td>%tid</td>
<td>%j</td>
<td>t₁</td>
<td>t₁</td>
<td>t₁</td>
<td>t₁</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>gt</td>
<td>%p₁</td>
<td>%ixj</td>
<td>%tid</td>
<td>t₂</td>
<td>t₂</td>
<td>t₂</td>
<td>t₂</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>bra</td>
<td>%p₁</td>
<td></td>
<td>L₂</td>
<td>t₃</td>
<td>t₃</td>
<td>t₃</td>
<td>t₃</td>
</tr>
<tr>
<td>4</td>
<td>L₂</td>
<td>and</td>
<td>%t₁</td>
<td>%tid</td>
<td>%k</td>
<td>t₅</td>
<td>•</td>
<td>t₅</td>
<td>•</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td>eq</td>
<td>%p₂</td>
<td>%t₁</td>
<td>0</td>
<td>t₆</td>
<td>•</td>
<td>t₆</td>
<td>•</td>
</tr>
<tr>
<td>6</td>
<td></td>
<td>bra</td>
<td>%p₂</td>
<td></td>
<td>L₃</td>
<td>t₇</td>
<td>•</td>
<td>t₇</td>
<td>•</td>
</tr>
<tr>
<td>7</td>
<td>L₃</td>
<td>load</td>
<td>%t₂</td>
<td>%shared</td>
<td>%tid</td>
<td>t₁₄</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>8</td>
<td></td>
<td>load</td>
<td>%t₃</td>
<td>%shared</td>
<td>%ixj</td>
<td>t₁₅</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>9</td>
<td></td>
<td>gt</td>
<td>%p₃</td>
<td>%t₂</td>
<td>%t₃</td>
<td>t₁₆</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>10</td>
<td></td>
<td>bra</td>
<td>%p₃</td>
<td></td>
<td>L₇</td>
<td>t₁₇</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>11</td>
<td>L₄</td>
<td>load</td>
<td>%t₄</td>
<td>%shared</td>
<td>%tid</td>
<td>•</td>
<td>•</td>
<td>t₈</td>
<td>•</td>
</tr>
<tr>
<td>12</td>
<td></td>
<td>load</td>
<td>%t₅</td>
<td>%shared</td>
<td>%ixj</td>
<td>•</td>
<td>•</td>
<td>t₉</td>
<td>•</td>
</tr>
<tr>
<td>13</td>
<td></td>
<td>lt</td>
<td>%p₄</td>
<td>%t₄</td>
<td>%t₅</td>
<td>•</td>
<td>•</td>
<td>t₁₀</td>
<td>•</td>
</tr>
<tr>
<td>14</td>
<td></td>
<td>bra</td>
<td>%p₃</td>
<td></td>
<td>L₇</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>t₁₁</td>
</tr>
<tr>
<td>15</td>
<td>L₅</td>
<td>store</td>
<td>%tid</td>
<td>%t₃</td>
<td>t₁₈</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td></td>
<td>store</td>
<td>%tid</td>
<td>%t₂</td>
<td>t₁₉</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td></td>
</tr>
<tr>
<td>17</td>
<td>L₆</td>
<td>store</td>
<td>%tid</td>
<td>%t₅</td>
<td>•</td>
<td>•</td>
<td>t₁₂</td>
<td>•</td>
<td></td>
</tr>
<tr>
<td>18</td>
<td></td>
<td>store</td>
<td>%tid</td>
<td>%t₄</td>
<td>•</td>
<td>•</td>
<td>t₁₃</td>
<td>•</td>
<td></td>
</tr>
<tr>
<td>19</td>
<td>L₇</td>
<td>sync</td>
<td></td>
<td></td>
<td>t₄</td>
<td>t₄</td>
<td>t₄</td>
<td>t₄</td>
<td></td>
</tr>
</tbody>
</table>
__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];
    }
}
Can you improve this code?

```
L2  %t1 = and %tid %k
    %p2 = eq %t1 0
    bra %p2 L3

7,329,816 / 28,574,321

L3  %t2 = ld %shared[%tid]
    %t3 = ld %shared[%ixj]
    %p3 = gt %t2 %t3
    bra %p3 L7

15,403,445 / 20,490,780

L4  %t4 = ld %shared[%tid]
    %t5 = ld %shared[%ixj]
    %p4 = lt %t4 %t5
    bra %p4 L7

4,651,153 / 8,083,541

L5  st %shared[%tid] %t3
    st %shared[%ixj] %t2

18

L6  st %shared[%tid] %t5
    st %shared[%ixj] %t4

12

sync

4
```
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]);
    }
}

• How can we improve this program to mitigate the problem of divergences?
First Optimization: 6.7% speed up*

(a)

```c
unsigned int a, b;
if ((tid & k) == 0){
  b = tid;
  a = ixj;
} else {
  b = ixj;
  a = tid;
}
if (sh[b] > sh[a]){  
  swap(sh[b],sh[a]);
}
```

(b)

```c
%t1 = and %tid %k
%p2 = eq %t1 0
bra %p2 L3

%b = mov %tid
%a = mov %ixj

%t2 = ld %shared[%b]
%t3 = ld %shared[%a]
%p3 = gt %t2 %t3
bra %p3 L7

st %shared[%a] %t3
st %shared[%b] %t2

sync
```

* Numbers refer to GTX 260
First Optimization: 6.7% speed up*

(a)

```c
unsigned int a, b;
if (((tid & k) == 0) {
    b = tid;
    a = ixj;
} else {
    b = ixj;
    a = tid;
}
if (sh[b] > sh[a]){
    swap(sh[b],sh[a]);
}
```

(b)

```
L2 %t1 = and %tid %k
%p2 = eq %t1 0
bra %p2 L3

L3 %b = mov %tid
%a = mov %ixj
bra %p2 L3/4

L3/4 %t2 = ld %shared[%b]
%t3 = ld %shared[%a]

L4 %b = mov %ixj
%a = mov %tid

L5/6 st %shared[%a] %t3
st %shared[%b] %t2

L7 sync
```

* Numbers refer to GTX 260

Can you do better?
Second Optimization: 9.2% speed up

(a)

```c
int p = (tid & k) == 0;
unsigned b = p?tid:ixj;
unsigned a = p?ixj:tid;

if (sh[b] > sh[a]) {
    swap(sh[b], sh[a]);
}
```

(b)

```
%t1 = and %tid %k
%p = eq %t1 0
%a = sel %tid %ixj %p
%b = sel %ixj %tid %p
%t2 = ld %shared[%b]
%t3 = ld %shared[%a]
%p3 = gt %t2 %t3
bra %p3 L7
```

```
L2
%t1 = and %tid %k
%p = eq %t1 0
%a = sel %tid %ixj %p
%b = sel %ixj %tid %p
%t2 = ld %shared[%b]
%t3 = ld %shared[%a]
%p3 = gt %t2 %t3
bra %p3 L7
```

```
L5/6
st %shared[%a] %t3
st %shared[%b] %t2
```

```
L7 sync
```
The final result

```c
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]);
    }
}
```

```c
int p = (tid & k) == 0;
unsigned b = p ? tid : ixj;
unsigned a = p ? ixj : tid;
if (shared[b] > shared[a]) {
    swap(shared[b], shared[a]);
}
```
Divergence Analysis

Given a program, which branches might cause divergences, and which branches will never do it?

- This analysis helps us to find which branches the compiler must optimize.
Why not to use profiling?

• Input dependent.
• Relatively hard to use.
• Huge slow-down: 140 times!
• The analysis can be easily embedded into the compiler.
• The analysis is definitive: if it says a branch is not divergent, then this branch will never diverge.
Our Solution: Divergence Analysis

A local variable is divergent if different threads see it with different values.

• Which variables are divergent?
Our Solution: Divergence Analysis

A local variable is divergent if different threads see it with different values.

• Which variables are divergent?
  – \( \nu = \text{tid} \)
  – \text{atomic}\{ \nu = f(...) \}
  – \nu \text{ is data dependent} on divergent variable \( u \).
  – \nu \text{ is control dependent} on divergent variable \( u \).
The thread id is always divergent

__global__
void saxpy (int n, float alpha, float *x, float *y) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    if (i < n) y[i] = alpha * x[i] + y[i];
}

Each thread sees a different thread id, so...

- But threads in different blocks may have the same threadIdx.x. Is this a problem?
Variables defined by atomic ops

• What is the program below doing?
  – The macro ATOMINC increments a global memory position, and returns the value of the result.

```c
__global__
void ex_atomic (int index, float* v) {
    int i = 0;
    i = ATOMINC( v[index] );
}
```
Static Single Assignment Form

• An intermediate representation in which each variable is defined only once.
  – We need to understand phi-functions.

\[
\begin{align*}
%b &= \text{mov} \%\text{tid} \\
%a &= \text{mov} \%\text{ixj} \\
%t2 &= \text{ld} \%\text{shared}[%b] \\
%t3 &= \text{ld} \%\text{shared}[%a] \\
%p3 &= \text{gt} \%t2 \%t3 \\
\text{bra} \%p3 \text{ L7}
\end{align*}
\]  

\[
\begin{align*}
%b &= \text{mov} \%\text{ixj} \\
%a &= \text{mov} \%\text{tid} \\
%b &= \text{mov} \%\text{ixj} \\
%a &= \text{mov} \%\text{tid} \\
%t2 &= \text{ld} \%\text{shared}[%b] \\
%t3 &= \text{ld} \%\text{shared}[%a] \\
%p3 &= \text{gt} \%t2 \%t3 \\
\text{bra} \%p3 \text{ L7}
\end{align*}
\]  

(a)  

\[
\begin{align*}
%b1 &= \text{mov} \%\text{tid} \\
%a1 &= \text{mov} \%\text{ixj} \\
%b2 &= \text{mov} \%\text{ixj} \\
%a2 &= \text{mov} \%\text{tid} \\
%a &= \phi (\%a1, \%a2) \\
%b &= \phi (\%b1, \%b2) \\
%t2 &= \text{ld} \%\text{shared}[%b] \\
%t3 &= \text{ld} \%\text{shared}[%a] \\
%p3 &= \text{gt} \%t2 \%t3 \\
\text{bra} \%p3 \text{ L7}
\end{align*}
\]  

(b)
What is Data Dependence?

• If the program contains an assignment such as \( v = f(v_1, v_2, ..., v_n) \), then \( v \) is data dependent on every variable in the right side, i.e, \( v_1, v_2, ..., \) and \( v_n \).
Which data dependences do we have here?
Divergences due to data dependences

Why is SSA so good here? Are we missing anything?
Can variable j diverge?

\[
%i0 = ld v[\%tid] \\
%j0 = 0
\]

\[
%i = \phi(%i0, %i1) \\
%j = \phi(%j0, %j3) \\
%p0 = %i < 100 \\
\text{branch } %p0 \ B_2
\]

\[
%i1 = %i + 1 \\
%j1 = %j + 1 \\
%t0 = %j1 \ mod \ 2 \\
%p1 = %t0 = 0 \\
\text{branch } %p1 \ B_4
\]

\[
%j2 = %j1 - 3
\]

\[
%j3 = \phi(%j2, %j1) \\
\text{sync} \\
\text{st } v[\%tid] \ %x0 \\
\text{stop}
\]

\[
%B_0\rightarrow B_1 \\
B_5 \rightarrow \text{sync} \\
%p2 = %j > 100 \\
\text{branch } %p2 \ B_7
\]

\[
%B_6 \rightarrow \%x0 = 1 \\
\text{jump } B_8
\]

\[
%B_7 \rightarrow \%x1 = 2
\]

\[
%B_8 \rightarrow \%x = \phi(%x0, %x1) \\
\text{sync} \\
\text{st } v[\%tid] \ %x0 \\
\text{stop}
\]

\[
%B_2 \rightarrow \text{sync} \\
%B_4 \rightarrow \text{jump } B_1
\]
Can variable x diverge?
Hard question: Can variable p1 diverge?

%i0 = ld v[%tid]
%j0 = 0

%i = \phi(%i0, %i1)
%j = \phi(%j0, %j3)
%p0 = %i < 100
branch %p0 B2

%i1 = %i + 1
%j1 = %j + 1
%t0 = %j1 mod 2
%p1 = %t0 = 0
branch %p1 B4

%j2 = %j1 - 3

%x0 = 1
jump B8

%x = \phi(%x0, %x1)
sync
st v[%tid] %x0
stop

%j3 = \phi(%j2, %j1)
sync
jump B1

B0
B1
B2
B3
B4
B5
B6
B7
B8
Sync Dependences

• A variable $v$ is sync dependent on a predicate variable $p$ if $v$ may reach a synchronization point with a different value for different threads, depending on how threads branch on $p$.

• How to find the set of sync dependent variables?

```
B_5
sync
%p2 = %j > 100
branch %p2 B_7

B_6
%x0 = 1
jump B_8

B_7
%x1 = 2

B_8
%x = $\phi(\%x0, \%x1)$
sync
st %v[%tid] \%x0
stop
```
The influence region

• The influence region of a predicate is the set of basic blocks that may (or may not) be reached depending on the value of the predicate, up to the synchronization barrier.

• The Synchronization barrier is placed at the *immediate post-dominator* of the branch.

• **Alert!** Head is spinning: what the heck is a post-dominator?
A Basic block \( B_1 \) post-dominates another basic block \( B_2 \) if, and only if, every path from \( B_2 \) to the end of the program goes across \( B_1 \).

For instance, in our running example, \( B_4 \) post-dominates \( B_2 \), thus, there is a sync barrier at \( B_4 \) to synchronize the threads that may diverge at \( B_2 \).
What is the influence region of p2?
The influence region of p2:

\( B_0 \):
\[
\begin{align*}
\%i_0 &= \text{l}d \ v[\%tid] \\
\%j_0 &= 0
\end{align*}
\]

\( B_1 \):
\[
\begin{align*}
\%i &= \phi(\%i_0, \%i_1) \\
\%j &= \phi(\%j_0, \%j_3) \\
\%p_0 &= \%i < 100 \\
\text{branch } \%p_0 &\ B_2
\end{align*}
\]

\( B_2 \):
\[
\begin{align*}
\%i_1 &= \%i + 1 \\
\%j_1 &= \%j + 1 \\
\%t_0 &= \%j_1 \mod 2 \\
\%p_1 &= \%t_0 = 0 \\
\text{branch } \%p_1 &\ B_4
\end{align*}
\]

\( B_3 \):
\[
\%j_2 &= \%j_1 - 3
\]

\( B_4 \):
\[
\begin{align*}
\%j_3 &= \phi(\%j_2, \%j_1) \\
\text{sync} \\
\text{st } \%v[\%tid] \%x_0 \\
\text{stop}
\end{align*}
\]

\( B_5 \):
\[
\begin{align*}
\text{sync} \\
\%p_2 &= \%j > 100 \\
\text{branch } \%p_2 &\ B_7
\end{align*}
\]

\( B_6 \):
\[
\%x_0 &= 1 \\
\text{jump } &\ B_8
\]

\( B_7 \):
\[
\%x_1 &= 2
\]

\( B_8 \):
\[
\begin{align*}
\text{sync} \\
\phi(\%x_0, \%x_1)
\end{align*}
\]

\( B_9 \):
\[
\begin{align*}
\%j_3 &= \phi(\%j_2, \%j_1) \\
\text{sync} \\
\text{jump } &\ B_1
\end{align*}
\]
What is the influence region of p1?
`%i0 = ld v[%tid]`  

`%j0 = 0`  

`%i = !(%i0, %i1)`  

`%j = !(%j0, %j3)`  

`%p0 = %i < 100`  

`branch %p0 B2`  

`%i1 = %i + 1`  

`%j1 = %j + 1`  

`%t0 = %j1 mod 2`  

`%p1 = %t0 = 0`  

`branch %p1 B4`  

`%j2 = %j1 - 3`  

`%x0 = 1`  

`jump B8`  

`%x1 = 2`  

`%x = !(%x0, %x1)`  

`sync`  

`st v[%tid] %x0`  

`stop`  

`The influence region of p1:`  

`%p2 = %j > 100`  

`branch %p2 B7`  

`%x = !(%x0, %x1)`  

`sync`  

`st v[%tid] %x0`  

`stop`
What is the influence region of p0?
The influence region of p0:

\begin{align*}
\%i &= \text{ld } v[\%tid] \\
\%j_0 &= 0
\end{align*}

\begin{align*}
\%i &= \phi(\%i_0, \%i_1) \\
\%j &= \phi(\%j_0, \%j_3) \\
\%p_0 &= \%i < 100 \\
\text{branch } \%p_0 & \rightarrow B_2
\end{align*}

\begin{align*}
\%i_1 &= \%i + 1 \\
\%j_1 &= \%j + 1 \\
\%t_0 &= \%j_1 \mod 2 \\
\%p_1 &= \%t_0 = 0 \\
\text{branch } \%p_1 & \rightarrow B_4
\end{align*}

\begin{align*}
\%j_2 &= \%j_1 - 3 \\
\%j_3 &= \phi(\%j_2, \%j_1) \\
\text{sync} \\
\text{st } v[\%tid] \%x_0 \\
\text{stop}
\end{align*}
The law of sync dependences

• **Theorem:** Let \textit{branch} \ %p \ B be a branch instruction, and let \( lp \) be its synchronization point. A variable \( v \) is sync dependent on \%p if, and only if, \( v \) is defined inside the influence region of the branch and \( v \) \textit{reaches} \( lp \).

  – Again: \( lp \) is the post-dominator of the place where the branch is defined.

  – **And alert again:** what does it mean for a variable to \textit{reach} a certain program point?
Which variables defined inside IR(%p0) reach its sync point?
How to transform sync dependences into data dependences?

• We augment phi-functions with predicates. For instance, to create a data dependence between \%x and \%p2:

\begin{align*}
\text{sync} & \quad \%p2 = \%j > 100 \\
\text{branch} & \quad \%p2 \\
\text{sync} & \quad \%p2 = \%j > 100 \\
\text{branch} & \quad \%p2
\end{align*}
How to transform sync dependences into data dependences?

- If a variable is alive outside the IR(%p), then we split its live range, with a unary phi-function.
%i0 = ld v[%tid]
%j0 = 0

%i = \phi(%i0, %i1)
%j = \phi(%j0, %j3)
%p0 = %i < 100
branch %p0 B2

%i1 = %i + 1
%j1 = %j + 1
%t0 = %j1 mod 2
%p1 = %t0 = 0
branch %p1 B4

%j2 = %j1 - 3

%x0 = 1
jump B8

%x = \phi(%x0, %x1), %p2
sync
st %v[%tid] %x0
stop

%B5
%j4 = \phi(%j), %p0
sync
%p2 = %j4 > 100
branch %p2 B7

%B6
%j4 = \phi(%j), %p0
sync
%p2 = %j4 > 100
branch %p2 B7

%B7
%x1 = 2

%B8
%j2 = %j1 - 3
As a matter of fact, this new program representation has a name:

GATED STATIC SINGLE ASSIGNMENT FORM
Sync dependence = data dependence
Sync Dependences Change Everything

OBSOLETE
Great! We know which branches are divergent. **And so what?**

- Thread re-location.
- Variable sharing.
- Barrier elimination.
- Branch fusion.

Let developers worry about their algorithms, while the **compiler** takes care of divergences.
I dream about a world without barriers

“All control constructs are assumed to be divergent points unless the control-flow instruction is marked as uniform, using the uni suffix. Therefore, a compiler or code author targeting PTX can ignore the issue of divergent threads, but has the opportunity to improve performance by marking branch points as uniform when the compiler or author can guarantee that the branch point is non-divergent.”

PTX programmer's manual
fernando@dcc.ufmg.br

%i0 = ld v[%tid]
%j0 = 0

%i = ! (%i0, %i1)
%j = ! (%j0, %j3)
%p0 = %i < 100
branch %p0 B2

%i1 = %i + 1
%j1 = %j + 1
%t0 = %j1 mod 2
%p1 = %t0 = 0
branch.uni %p1 B4

%j2 = %j1 - 3

%x0 = 1
jump B8

%x = ! (%x0, %x1)
sync
st %v[%tid] %x0
stop

%p2 = %j > 100
branch %p2 B7

%x1 = 2
jump B8

%x = ! (%x0, %x1)
sync
st %v[%tid] %x0
stop

%j3 = ! (%j2, %j1)
sync
jump B1

branch.uni %p1 B4
Variable Sharing

• The number of threads running on the GPU is bounded by the number of registers. E.g:
  – The GTX 8800 has 8,192 registers, and can run up to 768 threads. How many registers can we assign to each thread to get maximum thread usage?

If necessary, send non-divergent variables to the shared memory to make room for registers.
• Have you seen this kernel before?

```c
__global__ void regPress1(float* In, float* Out, int Width) {
    int tid = blockIdx.x * blockDim.x + threadIdx.x;
    if (tid < Width) {
        Out[tid] = 0.0;
        float a = 2.0F, b = 3.0F, c = 5.0F, d = 7.0F;
        for (int k = tid; k < Width * Width; k += Width) {
            Out[tid] += In[k] / (a - b);
            Out[tid] -= In[k] / (c - d);
            float aux = a;
            a = b;
            b = c;
            c = d;
            d = aux;
        }
    }
}
```

• Which variables are non-divergent?
• Do you remember what we can do with these variables?

```c
__global__ void regPress1(float* In, float* Out, int Width) {
    int tid = blockIdx.x * blockDim.x + threadIdx.x;
    if (tid < Width) {
        Out[tid] = 0.0;
        float a = 2.0F, b = 3.0F, c = 5.0F, d = 7.0F;
        for (int k = tid; k < Width * Width; k += Width) {
            Out[tid] += In[k] / (a - b);
            Out[tid] -= In[k] / (c - d);
            float aux = a;
            a = b;
            b = c;
            c = d;
            d = aux;
        }
    }
}
```

From `regPress`, available in the course webpage.
• This is not always worth it. What are the costs of sending data to shared memory?

```c
__global__ void regPress2(float* In, float* Out, int Width) {
    int tid = blockIdx.x * blockDim.x + threadIdx.x;
    if (tid < Width) {
        __shared__ float common0, common1;
        float c = 5.0F; float d = 7.0F;
        if (threadIdx.x == 0) { common0 = 2.0F; common1 = 3.0F; }
        __syncthreads();
        Out[tid] = 0.0;
        for (int k = tid; k < Width * Width; k += Width) {
            Out[tid] += In[k] / (common0 - common1);
            Out[tid] -= In[k] / (c - d);
            float aux = common0;
            if (threadIdx.x == 0) { common0 = common1; common1 = c; }
            __syncthreads();
            c = d; d = aux;
        }
    }
}
```

From `regPress`, available in the course webpage.
What the heck is Branch Fusion?

Find the longest sequences of common instructions in the two paths of a branch, and merge them.

- Branch fusion finds common work for lefties and righties, bringing more harmony to parallel.
Let’s consider an example:

```c
__global__ void exampleKernel
(float* u, float* v, float c1, float c2) {
  float y = u[tid];
  if (y != 0.0) {
    float x = v[tid];
    v[tid] = c1*x*x*x*y*y + c2*x;
  } else {
    float x = v[tid];
    v[tid] = c1*x*x*2.0 + c2*x*x;
  }
}
```

What is the longest chain of common instructions?

1. `%t0 = %u[%tidx]`
2. `%p2 = ne %t1 0.0
3. `bra %p2 (12)`
4. `%t9 = %v[%tidx]`
5. `%t10 = %t9 * %t9`
6. `%t11 = %t10 * %c1`
7. `%t12 = %t11 / 2.0`
8. `%t13 = %t9 * %c2`
9. `%t14 = %t13 * %t9`
10. `%t15 = %t12 + %t14`
11. `%v[%tidx] = %t8`
How do we do branch fusion?

• The answer comes from *computational biology*:
  – **Smith-Waterman** sequence alignment.
  – But, instead of *genes*, we match *instructions*. 
Let’s try to align the instructions:

$$T = \begin{array}{cccccccc}
\checkmark & \checkmark & \checkmark & X & X & \checkmark & X & \checkmark & \checkmark & \checkmark & \checkmark \\
\downarrow & * & * & \_ & \_ & \_ & / & * & * & + & \uparrow \\
\end{array}$$

$$F = \begin{array}{cccccccc}
\downarrow & * & * & * & / & / & \_ & \_ & * & + & \uparrow \\
\end{array}$$

How to find these chains automatically?
Smith-Waterman
The Profitability Matrix

- Each cell is the profit of merging two instructions.
- What is the profit of merging two divisions?
- What represents the most profitable path inside the matrix?
- What are the diagonals in this path?
- What are the vertical and horizontal paths?
  - Is there a cost in leaving a diagonal?
  - Is there a cost in entering a diagonal?
Touché!

```plaintext
1  %t0 = %u[%tidx]
2  %p2 = ne %t1 0.0
3  bra %p2 (12)

12 %t1 = %v[%tidx]
13 %t2 = %t1 * %t1
14 %t3 = %t2 * %t1
15 %t4 = %t3 * %c1
16 %t5 = %t4 / %t0
17 %t6 = %t5 / %t0
18 %t7 = %t1 * %c2
19 %t8 = %t6 + %t7
20 %v[%tidx] = %t8

4  %t9 = %v[%tidx]
5  %t10 = %t9 * %t9
6  %t11 = %t10 * %c1
7  %t12 = %t11 / 2.0
8  %t13 = %t9 * %c2
9  %t14 = %t13 * %t9
10 %t15 = %t12 + %t14
11 %v[%tidx] = %t8

1  %t0 = %u[%tidx]
2  %p2 = ne %t1 0.0
3  bra %p2 (13)

12 %t1 = %v[%tidx]
13 %t4 = %t3 * %c1
14 %t5 = %t4 / %t0
15 %t14 = %t7_13 * %t1_9
16 %t8_15 = %t6_12 + %t7_13
17 %v[%tidx] = %t8_15
```
The matrix and the program
Data Re-location

• Sometimes it pays off spending some time to reorder the data before processing it.
  — Or then we can try to “quasi-sort” these data.

• What could we do to improve the execution of this old acquaintance of ours?

```c
__global__ void dec2zero(int* v, int N) {
    int xIndex = blockIdx.x*blockDim.x+threadIdx.x;
    if (xIndex < N) {
        while (v[xIndex] > 0) {
            v[xIndex]--;
        }
    }
}
```
Quasi-sorting

• Copy data to shared memory
• Each thread sorts small chunks of data, say, four cells
• Scatter the data around the shared memory:
  – Each thread puts its smallest element in the first $\frac{1}{4}$ of the array, the second smallest in the second $\frac{1}{4}$ of the array, and so forth.
The Rainbow’s End

• Many conferences take papers on optimization of GPU programs:
  – PLDI, POPL, PPoPP, PACT, MICRO, ASPLOS
• There are many discussion forums
  – Nvidia, Ocelot
• Books
  – Computer Organization and Design
  – Lots of Nvidia on-line books