GPUs are important!

- high-throughput
- scalable
- deep learning
if (tid + k+1) >= N) return;
x = tid + k+1;

for (y=k; y < N; y++){
    xk = N * x + k;
    xy = N * x + y;
    ky = N * k + y;
    if (y == k)
        B[x] -= M[xk] * B[k];
}

Kernel

Threads

Single Instruction Multiple Threads

local

global
if(tid+t+1 >= N) return;
x = tid+t+1;

for(y=t; y < N; y++){
    xt = N*x + t;
    xy = N*x + y;
    ty = N*t + y;
    if(y == t)
        B[x] -= M[xt]*B[t];
}
Warp scheduling

<table>
<thead>
<tr>
<th>Instructions</th>
<th>Warps</th>
</tr>
</thead>
<tbody>
<tr>
<td>i0</td>
<td>w0</td>
</tr>
<tr>
<td>i1</td>
<td>w1</td>
</tr>
<tr>
<td>i2</td>
<td>w2</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

Round-robin scheduling
Uncoalesced Global Memory Accesses (performance bug)

warp w0

tid: 0 1 2 .... 31

load A[x]

×: 0 0 0 .... 0

1 global memory transaction

×: 0 1 2 .... 31

1 global memory transaction

×: 0 8 16 .... 248

8 global memory transactions

Uncoalesced
Compiler Approaches for Performance

• Data layout transformation
  Che et al. (2011), Zhang et al. (2011), Sung et al. (2010), Jang et al. (2011)

• Loop/threads restructuring
  Yang et al. (2010), Baskaran et al. (2008)

• Caching data programatically
  Ueng et al. (2008), Yang et al. (2010), Baskaran et al. (2008), Jang et al. (2011)
Hardware Approaches for Performance

- **Schedule warps in small groups**
  Narasiman et al. (2011), Jog et al. (2013), Gong et al. (2017)

- **Restrict number of warps scheduled**
  Rogers et al. (2012), Sethia et al. (2015)

- **Bypass cache**
  Jia et al. (2012, 2014), Xie et al. (2013)
Drawbacks

• Compiler approaches
  • Complex transformations
  • Easy to make errors

• Hardware approaches
  • Limited control state per thread
  • Narrow view of the execution
Proposal

- Static Analysis
  - to detect performance bugs in GPU programs
  - to identify performance opportunities in GPU hardware
  - to perform correct GPU program transformations
Proposal

• Static Analysis

• to detect performance bugs in GPU programs

• to identify performance opportunities in GPU hardware

• to verify GPU program transformations to fix performance bugs
GPUDrano

• Uncoalesced global memory accesses
  • accesses that require more than 1 transaction
  • dynamic property

• Amenable to static analysis
  • access indices exhibit regular static patterns
Dependence of $\text{tid}$

- Consider a load on $A[x]$
  - $x = \text{constant} : \text{coalesced}$
  - $x = \text{tid} + \text{constant} : \text{coalesced}$
  - $x = -\text{tid} + \text{constant} : \text{coalesced}$
  - otherwise : uncoalesced
Example

```c
if(tid + k+1 >= N) return;

x = tid + k+1;

for(y=k; y < N; y++){
    xk = N * x + k;
    xy = N * x + y;
    ky = N * k + y;


    if(y == k)
        B[x] -= M[xk] * B[k];
}
```

Gaussian Elimination Kernel
Execution for a warp

```c
if(tid + k+1 >= N) return;
x = tid + k+1;

for(y=k; y < N; y++){
xk = N * x + k;
xy = N * x + y;
ky = N * k + y;

if(y == k)
  B[x] -= M[xk] * B[k];
}
```

Accesses $A[xy]$ and $M[xk]$ are uncoalesced!
Example - static analysis

```c
if(tid + k+1 >= N) return;
x = tid + k+1;

for(y=k; y < N; y++){
  xk = N * x + k;
  xy = N * x + y;
  ky = N * k + y;

  if(y == k)
    B[x] -= M[xk] * B[k];
}

k → 0
N → 0
x → 1 + 0 = 1
y → 0
xk → 0*1 + 0 = T + 0 = T
xy → 0*1 + 0 = T + 0 = T
ky → 0*0 + 0 = 0

0*1 = (c)*(tid + c) = T
```
Boolean Abstraction

• What if an access is executed for a single thread in the warp?

```plaintext
x = tid
if(x == 5){
y = A[N * x + 1]
}
```

executed only for `tid = 5`

• Boolean abstraction

• Dependence of boolean predicates on `tid`

• `tf(c), t-(tid = c), f-(tid != c), T(otherwise)`
Static Analysis

• Abstract Semantics defined for the abstraction

• Intra-procedural analysis

  • Variables initialised to 0, tf

  • Program executed using abstract semantics

  • Accesses with index T are labelled uncoalesced

• Loops - execution repeated until fixed point reached
Evaluation

• Implemented in gpucc, open-source CUDA compiler

• Compared with Dynamic Analysis implementation

• Evaluated on Rodinia benchmark suite
Evaluation: Recall

111 out of 143 real bugs caught by SA
Evaluation: Precision

111 out of 180 reported bugs by SA are real
Evaluation: Runtime

Takes less than 1 second for most benchmarks
Proposal

• Static Analysis

  • to detect performance bugs in GPU programs

• to identify performance opportunities in GPU hardware

• to perform correct GPU program transformations
Static Analysis based warp-scheduling (proposed work)

<table>
<thead>
<tr>
<th>Hardware analysis</th>
<th>Static analysis</th>
</tr>
</thead>
<tbody>
<tr>
<td>works at run-time</td>
<td>works at compile-time</td>
</tr>
<tr>
<td>narrow view of execution</td>
<td>abstraction for wider view</td>
</tr>
</tbody>
</table>

Use static analysis to identify performance improvement opportunities
Cross-iteration cache reuse

- Cache reuse
  - Reduces access latency significantly
- Identify cache reuse across iterations of a loop
  - Difficult in hardware, if the loop is large
Example

```
if(tid + k+1 >= N) return;
x = tid + k+1;

for(y=k; y < N; y++){
xk = N * x + k;
xy = N * x + y;
ky = N * k + y;
    if(y == k)
        B[x] -= M[xk] * B[k];
}
```

Execution for \textbf{tid} = 0
\begin{align*}
k &= 0, N = 32 \\
\text{Iteration } y &= 0 \\
xy &= 32 \\
\text{A[32] to A[63] in cache}
\text{Iteration } y &= 1 \\
xy &= 33 \\
\text{A[33] accessed}
\end{align*}
Static Analysis

• For an access \( A[x] \) in a loop
  
  • Compute \( x_1 \) and \( x_2 \) in consecutive iterations
  
  • If difference \( (x_1 - x_2) \) is within cache-line
    
    • Label access \( A[x] \) as reusable
Hardware Strategies

- Ensure that for reusable accesses
  - Fetched cache-data is not evicted before reuse
- Bypass cache for non-reusable accesses
- Restrict the number of warps scheduled
Proposal

• Static Analysis
  • to detect performance bugs in GPU programs
  • to identify performance opportunities in GPU hardware
  • to perform correct GPU program transformations
Verified Program Transformation Framework (proposed work)

• Need to verify transformations
  • Formal model of programs required
  • Formal specification of transformation required
• Framework to specify, implement and verify transformations
Example

```c
if(tid + k+1 >= N) return;
x = tid + k+1;

for(y=k; y < N; y++){
xk = N * x + k;
xy = N * x + y;
ky = N * k + y;

if(y == k)
    B[x] -= M[xk] * B[k];
}
```

// thread-parallel
for(t=0; t < M; t++){
    { 
        L1: if(t+k+1 >= N) return;
x = t+k+1;
    }
    for(y=k; y < N; y++){
        L2: xk = N * x + k;
            xy = N * x + y;
            ky = N * k + y;
        if(y == k)
            B[x] -= M[xk] * B[k];
    }
}

Formal model
// thread-parallel
for (t=0; t < M; t++){
    {  
        L1: if(t+k+1 >= N) return;
            x = t+k+1;
    }
    for(y=k; y < N; y++){
        L2: xk = N * x + k;
            xy = N * x + y;
            ky = N * k + y;
            if(y == k)
                B[x] -= M[xk] * B[k];
    }
}

t iterates over rows of A
y iterates over columns of A

Possible transformations to fix access A[xy]

• Interchange loops t and y
• Change layout of A to column-major
Interchange loops $t$ and $y$

// thread-parallel
for (t=0; t < M; t++){
    {  
        L1: if(t+k+1 >= N) return;
            x = t+k+1;
    }
    for(y=k; y < N; y++){
        L2: xk = N * x + k;
            xy = N * x + y;
            ky = N * k + y;
            if(y == k)
                B[x] -= M[xk] * B[k];
    }
}
Interchange loops $t$ and $y$

```c
// thread-parallel
for (t=0; t < M; t++){
  for (y=k; y <= k; y++){
    L1: if(t+k+1 >= N) return;
        x = t+k+1;
  }
  for(y=k; y < N; y++){
    L2: xk = N * x + k;
        xy = N * x + y;
        ky = N * k + y;
        if(y == k)
          B[x] -= M[xk] * B[k];
  }
}
```

$L = \text{split-merge}(L1, L2, 2)$
Interchange loops $t$ and $y$

// thread-parallel
for (t=0; t < M; t++){
  for (y=k; y < N; y++){
    L: if(y == k){
      if(t+k+1 >= N) return;
      x = t+k+1;
    }
    xk = N * x + k;
    xy = N * x + y;
    ky = N * k + y;

    if(y == k)
      B[x] -= M[xk] * B[k];
  }
}

L = split-merge(L1, L2, 2)
Interchange loops $t$ and $y$

// thread-parallel
for (t=0; t < M; t++){
    for (y=k; y < N; y++){
        L:  if(y == k){
            if(t+k+1 >= N) return;
            x = t+k+1;
        }
        xk = N * x + k;
        xy = N * x + y;
        ky = N * k + y;


        if(y == k)
            B[x] -= M[xk] * B[k];
    }
}

L = split-merge(L1, L2, 2)
affine(L, [tid, y] -> [y, tid])
Interchange loops \( t \) and \( y \)

```c
// thread-parallel
for (y=k; y < N; y++){
    for (t=0; t < M; t++)
    {
        L: if(y == k){
            if(t+k+1 >= N) return;
            x = t+k+1;
        }
        xk = N * x + k;
        xy = N * x + y;
        ky = N * k + y;


        if(y == k)
            B[x] -= M[xk] * B[k];
    }
}

L = split-merge(L1, L2, 2)
affine(L, [tid, y] -> [y, tid])
```

Verify resulting model is equivalent to original model
Interchange loops $t$ and $y$

```plaintext
if (y >= N || y < t) return;
for (t=0; t < M; t++){
    L:
      if (y == k){
        if (t+k+1 >= N) return;
        x = t+k+1;
      }
    xk = N * x + k;
    xy = N * x + y;
    ky = N * k + y;
    if (y == k)
      B[x] -= M[xk] * B[k];
  }
}

L = split-merge(L1, L2, 2)
affine(L, [tid, y] -> [y, tid])

Final program
```
Framework

• Formal model
  • Labelled loop nests/ parts of loop nests
  • Loop representing threads

• Transformations
  • split-merge, affine: loop restructuring
  • data-affine: data-layout transformation
  • data-shared: caching data programmatically
Plan

• Extend Polyhedral Model
  • Well-known formal model for programs
  • Formal model: layer on top of polyhedral model
  • Transformations: operations on polyhedral model
• Loopy (SAS 2016)
  • Similar framework for loop transformations on sequential programs
Timeline

• GPUDrano - completed (paper submitted at CAV’17)

• SA based warp-scheduling (April - Sept 2017)

• SA Internship @ Google (June - Aug 2017)

• Program Trans. Framework (Oct 2017 - March 2018)

• Dissertation (April - May 2018)
Conclusion

• Unique features of GPU programs
  • Large number of threads
  • Execution of threads is very similar
• Static Analysis to utilize this similarity
Thank you!