Correctly Treating Synchronizations in Compiling Fine-Grained SPMD-Threaded Programs for CPU

Ziyu Guo, Eddy Zheng Zhang and Xipeng Shen
Department of Computer Science
College of William and Mary

Heterogeneity

Write once, run everywhere FAST
Run everywhere fast: How?

• Cross platform translation/optimization

• CPU to GPU translation
  – Ayguade+, IWOMP’09: OpenMP tasking
  – Lee+, PPOPP’09: Translating OpenMP loop
  – Michel+, HIPS’11: SkelCL
  – Etc...

• GPU to CPU translation
  – Stratton+, LCPC’08 & CGO’10: MCUDA
  – Diamos+, PACT’10: OCELOT
  – Lee+, PACT’10: OpenCL framework

A largely overlooked problem

• Beware of hardware-specific features!

Synchronization mechanisms on GPGPUs
e.g. __syncthreads(), barrier(), locksteps, ...
SIMD on CUDA: an overview

- **Warp**
  - Data-level parallel
  - Single Instruction Multiple Data (SIMD)
  - Implicit synchronizations (locksteps) in a warp
- **Block**
  - Contains multiple warps
  - Task-level parallel
  - Explicit synchronizations (__syncthreads()) in a block

```
i=threadIdx.x;
A[i] = A[i-1];
__syncthreads();
A[i] = A[i+1];
```

State-of-Art GPU→ CPU Translation

- Explicit synchronizations→ loop fission
  - loop boundary = barrier in CPU thread loop iterations

Problem Showcase

• Dependences originally protected by imp syncs

A[i] = A[i-1];  
//GPU code

for(tid=0;tid<warpSize;tid++){  
  A[i] = A[i-1]  
}  
//MCUDA Generated CPU code

CPU Thread loops do not keep GPU access orders exactly!
Finding the Problem

A[i] = A[i-1];

//GPU code

for(tid=0;tid<warpSize;tid++){
    A[i] = A[i-1]
}

//MCUDA Generated CPU code

Finding the Problem

Directions of dependencies changed!

A[i] = A[i-1];

//GPU code

for(tid=0;tid<warpSize;tid++){
    A[i] = A[i-1]
}

//MCUDA Generated CPU code


Real-world Example

• Reduction kernel v5 from NVIDIA CUDA SDK

```c
// s[]: (volatile) input array
for (i=blockSize/2; i>=32; i>>=1){
    if (tid < i)    s[tid] += s[tid+i];
    __syncthreads();
}
if (tid < 32){
    s[tid] += s[tid+32]; __EMUSYNC;
    s[tid] += s[tid+16]; __EMUSYNC;
    s[tid] += s[tid+8];  __EMUSYNC;
    s[tid] += s[tid+4];  __EMUSYNC;
    s[tid] += s[tid+2];  __EMUSYNC;
    s[tid] += s[tid+1];
}
```

(a) GPU kernel function  (b) Algorithm

Goal

Correctly Handle Implicit Synchronizations
Outline

• Insufficiency of Simple Extensions

• SPMD Translation Dependence Theorem

• Merging and Splitting Oriented Solutions

• Evaluation

• Conclusion

A Simple MCUDA Extension

• Treats all implicit syncs as explicit ones.
  – MCUDA’s code generation + references extraction
  – Creates a separate loop for all such statements

• Relies on available compiler loop fusion capability
  – Limited performance on such deeply fissioned loops
  – Still cannot handle thread-dependent conditions
Thoughts: the Other Way Around

• Step 0: Have MCUDA generate the thread loops

• Step 1: Identify where implicit synchronizations might actually do damages

• Step 2: Insert syncs to protect ordering
  – Where there're actual dependences
  – Dependence analysis

Outline

• Insufficiency of Simple Extensions

• SPMD Translation Dependence Theorem

• Merging and Splitting Oriented Solutions

• Evaluation

• Conclusion
Terms for Next Few Slides

• **Sequence Number**
  – In Reverse-post ordered basic blocks

• **Dependence Distance Vector**
  – Diff in the iteration vectors when 2 statements access the same data entry.

```
for(tid=...){
    for(i=...){
        for (j=...){
            A[tid][j][i+2] = ...;
            ... = A[tid+1][j+1][i];
        }
    }
}
```

**DDV from S1 to S2: (tid, i, j) = (1, -2, 1)**

Terms for Next Few Slides

- **Non-critical Dependence**
  - Retains by basic transformations
  - As contrary to Critical Dependence

```
for(i=...){
    for (j=...){
        A[tid][j][i+2] = ...;
        ... = A[tid+1][j+1][i];
    }
}
```

// MCUDA generated CPU code

```
tid = blockIdx.x*blockSize+threadIdx.x;
for(i=...){
    for (j=...){
        A[tid][j][i+2] = ...;
        ... = A[tid+1][j+1][i];
    }
}
```

// GPU kernel

**What happens to A[1][2][3]??**

Written by S1 in thread 1 at (i=1, j=2)  Written by S1 at (tid=1, i=1, j=2)
Read by S2 in thread 0 at (i=3, j=1)  Read by S2 at (tid=0, i=3, j=1)

Critical dependence alert
Case Study 1

for(i=...){
  S1: A[tid][i] = ...;
  S2: ... = A[tid][i+1];
  //GPU
}

for(tid=...){
  for(i=...){
    S1: A[tid][i] = ...;
    S2: ... = A[tid][i+1];
  }
  //CPU
  //DDV [0, 1]

  Each thread "owns" 1 row, no cross-thread dependence at all.

Condition 1:
if DDV(1) == 0, there’s no inter-thread dependence.
MCUDA translation is safe.

Case Study 2

for(i=...){
  S1: A[tid+1][i] = ...;
  S2: ... = A[tid][i];
  //GPU
}

for(tid=...){
  for(i=...){
    S1: A[tid+1][i] = ...;
    S2: ... = A[tid][i];
  }
  //CPU
  //DDV [-1, 0]

  A[1][1] written by S1 in thread 0 at (i=1) then
  A[1][1] written by S1 at (tid=0, i=1) then
  A[1][1] read by S2 in thread 1 at (i=1) then
  A[1][1] read by S2 at (tid=1, i=1)

Condition 2:
if DDV(1) < 0, and the rest entries are all-0, inter-thread dependence is preserved in thread loop.
Case Study 3

Both “wavefronts” goes the “same direction” as the referencing order within threads

Both are Write After Read
Case Study 3

Condition 3:
If \( DDV(1) \) equals the first non-zero entry afterwards, the dependence is preserved in thread loop

```
for(i=...){
    A[tid][i] = ...;
    ... = A[tid+1][i+1];}
//GPU
```
```
for(tid=...){
    for(i=...){
        A[tid][i] = ...;
        ... = A[tid+1][i+1];}
//DDV [1, 1]
//CPU
```

SPMD Translation Dependence Theorem and Its Implications

Let \( S_1 \) and \( S_2 \) be two statements in \( L \) and \( sn(S_1) < sn(S_2) \).

Let \( d \) be a data dependence from \( S_1 \) to \( S_2 \) in \( C \). Let \( v \) be the sign vector of the data dependence in \( L \) that corresponds to \( d \). The dependence \( d \) is preserved in \( L \) if at least one of the following conditions holds:

1. \( v(1) == 0 \);
2. there are no non-zero element in \( v(2:|v|) \) & \( v(1) < 0 \);
3. \( v(1) \) equals the first non-zero element in \( v(2:|v|) \) and that element is not "*".
SPMD Translation Dependence Theorem and Its Implications

• Enables compiler examination of SPMD translation

• Enables compiler to locate critical dependences during translation

• Guides the compiler towards proper loop transformation to fix the errors

Outline

• Insufficiency of Simple Extensions

• SPMD Translation Dependence Theorem

• Merging and Splitting Oriented Solutions

• Evaluation

• Conclusion
Splitting-Oriented Solution

Apply MCUDA transformation on segments bounded by exp syncs

Compute all pairs of DDVs

Classify deps into 4 kinds:

- **Intra-thread**
  - Benign/Reversible
  - Critical

**If** \( R = C = \emptyset \)

**Y**

Insert additional syncs

**N**

Create separate thread loops (with strict ordering between each other) for different kinds of dependences

// W: warp size
for (i=W-1; i>=0; i--){
    S1: B[i] = ...
    S2: B[i+16] = ...
    S3: ... = A[i]
    S4: ... = A[i+16]
    if (...){
        for (i=0; i<W; i++){
            S5: A[i+8] = ...
        }
    }
    if (...){
        for (i=0; i<W; i++){
            S6: C[i+8] = ...
        }
    }
    for (i=0; i<W; i++){
        S7: C[i] = ...
    }
}
Merging-Oriented Solution

- Treats all implicit syncs as explicit ones.

- And
  - Predicate conditional statements
  - Code replication and bookkeeping to ensure each threads' private conditions are correct

```plaintext
for(tid=...){
  for(tid=...){
  }
  for(tid=...){
  }
}
```

Section Summary

- Step 1: Identify critical dependences
- Step 2: Insert syncs to force ordering
  - Further loop fission: more thread loops
  - Ascending index for Intra-thread/Benign dependences
  - Descending index for Reversible dependences
  - Separate loop for each Critical dependences
- Same for relaxing over-strict exp syncs
  - Warp scope --> Block scope
Outline

• Insufficiency of Simple Extensions
• SPMD Translation Dependence Theorem
• Merging and Splitting Oriented Solutions
• Evaluation
• Conclusion

Benchmarks and Versions

<table>
<thead>
<tr>
<th>Program Version</th>
<th>Description</th>
<th>Synchronization types</th>
</tr>
</thead>
<tbody>
<tr>
<td>Incorrect Baseline</td>
<td>MCUDA produced version</td>
<td></td>
</tr>
<tr>
<td>Merging Oriented</td>
<td>Simple extension to MCUDA</td>
<td></td>
</tr>
<tr>
<td>Splitting Oriented</td>
<td>With dependence analysis</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Program</th>
<th>Description</th>
<th>Synchronization types</th>
</tr>
</thead>
<tbody>
<tr>
<td>CG</td>
<td>Conjugate gradient</td>
<td>Explicit and Implicit</td>
</tr>
<tr>
<td>Reduction</td>
<td>Parallel reduction</td>
<td>Explicit and Implicit</td>
</tr>
<tr>
<td>SortingNetwork</td>
<td>Bitonic sort</td>
<td>Explicit and Implicit</td>
</tr>
<tr>
<td>sgemm</td>
<td>Dense matrix calculation</td>
<td>Explicit</td>
</tr>
<tr>
<td>transposeNew</td>
<td>Matrix transpose</td>
<td>Explicit</td>
</tr>
</tbody>
</table>
Speedups

Platforms

<table>
<thead>
<tr>
<th>Platforms</th>
<th>AMD Quadcore + GCC</th>
<th>Intel Quadcore+ICC</th>
</tr>
</thead>
</table>

Lower is better

Splitting oriented version guarantees correctness with reasonable overhead

ICC delivers better performance for merging, but still no match with splitting

Discussion

• Impact beyond cuda
  – Same problem exists in OpenCL
  – Dependence analysis always necessary

• Statement-level analysis vs. instance-level analysis
  – Statement-level synchronizations too strong a constraint
  – Relax this constraints by breaking the loop structure for better performance (LCPC’11)

• Warp-level intrinsics (__any(), __all())
  – Append warp-level barrier after each
Conclusion

• Contribution of this work
  – For the first time
    • Formalized statement-level analysis to identify critical dependences
    • GPU-to-CPU compilation with guaranteed correctness
  – Compared to previous approaches
    • Automatic
    • Basis for further analysis and optimization

Thanks!
Questions?