A General Purpose Automatic Overlapped Tiling Technique in Polyhedral Frameworks

Jie Zhao

INRIA & École Normale Supérieure
45 rue d’Ulm, 75005 Paris, France

ACM Student Research Competition (SRC)
2018 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)
Vienna, Austria

January 26, 2018
1 Background
2 Comparing with a DSL Polyhedral Framework
3 Polyhedral Implementation of the Overlapped Tiling
4 Experimental Results
5 Conclusion and Future Work
for (t = 0; t < T; t++)
    for (i = 1; i < N - 1; i++)
        A[t + 1][i] = 0.25 * (A[t][i + 1] + 2.0 * A[t][i] + A[t][i - 1]);
for (t=0; t<T; t++)
    for (i=1; i<N-1; i++)
        A[t+1][i] = 0.25*(A[t][i+1] + 2.0*A[t][i] + A[t][i-1]);

Stencil Code from heat-1d.c

Different tile shapes on the above stencil code’s iteration space
for (t=0; t<T; t++)
for (i=1; i<N-1; i++)
    A[t+1][i] = 0.25*(A[t][i+1] + 2.0*A[t][i] + A[t][i-1]);

Stencil Code from heat-1d.c

Different tile shapes on the above stencil code’s iteration space
for (t=0; t<T; t++)
  for (i=1; i<N-1; i++)
    A[t+1][i] = 0.25*(A[t][i+1]+2.0*A[t][i]+A[t][i-1]);

Stencil Code from heat-1d.c

Different tile shapes on the above stencil code’s iteration space
for (t=0; t<T; t++)
for (i=1; i<N-1; i++)
    A[t+1][i]=0.25*(A[t][i+1]+2.0*A[t][i]+A[t][i-1]);

Stencil Code from heat-1d.c
<table>
<thead>
<tr>
<th></th>
<th>parallelogram</th>
<th>diamond</th>
<th>split</th>
<th>overlapped</th>
</tr>
</thead>
<tbody>
<tr>
<td>shape complexity</td>
<td>×</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>parallelism</td>
<td>×</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>locality</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>redundancy</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>✓</td>
</tr>
<tr>
<td>algorithmic complexity</td>
<td>×</td>
<td>✓</td>
<td>✓</td>
<td>×</td>
</tr>
</tbody>
</table>
Comparison Between Different Tiling Techniques

<table>
<thead>
<tr>
<th></th>
<th>parallelogram</th>
<th>diamond</th>
<th>split</th>
<th>overlapped</th>
</tr>
</thead>
<tbody>
<tr>
<td>shape complexity</td>
<td>×</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>parallelism</td>
<td>×</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>locality</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>redundancy</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>✓</td>
</tr>
<tr>
<td>algorithmic complexity</td>
<td>×</td>
<td>✓</td>
<td>✓</td>
<td>×</td>
</tr>
</tbody>
</table>

- Our goal: implement an effective tiling technique for programs written in general purpose languages:
## Comparison Between Different Tiling Techniques

<table>
<thead>
<tr>
<th></th>
<th>parallelogram</th>
<th>diamond</th>
<th>split</th>
<th>overlapped</th>
</tr>
</thead>
<tbody>
<tr>
<td>shape complexity</td>
<td>×</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>parallelism</td>
<td>×</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>locality</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>redundancy</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>✓</td>
</tr>
<tr>
<td>algorithmic complexity</td>
<td>×</td>
<td>✓</td>
<td>✓</td>
<td>×</td>
</tr>
</tbody>
</table>

- Our goal: implement an effective tiling technique for programs written in general purpose languages:
  - Concurrent start—parallelogram tiling
## Comparison Between Different Tiling Techniques

<table>
<thead>
<tr>
<th></th>
<th>parallelogram</th>
<th>diamond</th>
<th>split</th>
<th>overlapped</th>
</tr>
</thead>
<tbody>
<tr>
<td>shape complexity</td>
<td>✗</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>parallelism</td>
<td>✗</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>locality</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>redundancy</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✓</td>
</tr>
<tr>
<td>algorithmic complexity</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✗</td>
</tr>
</tbody>
</table>

- Our goal: implement an effective tiling technique for programs written in general purpose languages:
  - Concurrent start—**parallelogram tiling**
  - Different target architectures (CPUs and GPUs)—**diamond tiling**
## Comparison Between Different Tiling Techniques

<table>
<thead>
<tr>
<th></th>
<th>parallelogram</th>
<th>diamond</th>
<th>split</th>
<th>overlapped</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>shape complexity</strong></td>
<td>×</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td><strong>parallelism</strong></td>
<td>×</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td><strong>locality</strong></td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td><strong>redundancy</strong></td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>✓</td>
</tr>
<tr>
<td><strong>algorithmic complexity</strong></td>
<td>×</td>
<td>✓</td>
<td>✓</td>
<td>×</td>
</tr>
</tbody>
</table>

Our goal: implement an effective tiling technique for programs written in general purpose languages:

- Concurrent start—parallelogram tiling
- Different target architectures (CPUs and GPUs)—diamond tiling
- Redundancy vs. algorithmic complexity—split tiling
<table>
<thead>
<tr>
<th></th>
<th>parallelogram</th>
<th>diamond</th>
<th>split</th>
<th>overlapped</th>
</tr>
</thead>
<tbody>
<tr>
<td>shape complexity</td>
<td>×</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>parallelism</td>
<td>×</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>locality</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>redundancy</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>✓</td>
</tr>
<tr>
<td>algorithmic complexity</td>
<td>×</td>
<td>✓</td>
<td>✓</td>
<td>×</td>
</tr>
</tbody>
</table>

Our goal: implement an effective tiling technique for programs written in general purpose languages:

- Concurrent start—parallelogram tiling
- Different target architectures (CPUs and GPUs)—diamond tiling
- Redundancy v.s. algorithmic complexity—split tiling
- Overlapped tiling
Comparing with PolyMage

for (i=1; i<N; i++)
    A[i]=f(i);
for (i=2; i<N-1; i++)
for (i=4; i<N-3; i++)
    C[i]=0.25*(B[i-2]+2*B[i]+B[i+2]);

A simple image processing pipeline

Overlapped tiling on the iteration space
Comparing with PolyMage

for (i=1; i<N; i++)
    A[i]=f(i);
for (i=2; i<N-1; i++)
for (i=4; i<N-3; i++)
    C[i]=0.25*(B[i-2]+2*B[i]+B[i+2]);

A simple image processing pipeline

Overlapped tiling on the iteration space
Comparing with PolyMage

```c
for (i=1; i<N; i++)
    A[i]=f(i);
for (i=2; i<N-1; i++)
for (i=4; i<N-3; i++)
    C[i]=0.25*(B[i-2]+2*B[i]+B[i+2]);
```

A simple image processing pipeline

Overlapped tiling on the iteration space
Comparing with PolyMage

```c
for (i=1; i<N; i++)
    A[i]=f(i);
for (i=2; i<N-1; i++)
for (i=4; i<N-3; i++)
    C[i]=0.25*(B[i-2]+2*B[i]+B[i+2]);
```

A simple image processing pipeline

Overlapped tiling on the iteration space
Comparing with PolyMage

for (i = 1; i < N; i++)
    A[i] = f(i);
for (i = 2; i < N - 1; i++)
for (i = 4; i < N - 3; i++)
    C[i] = 0.25 * (B[i - 2] + 2 * B[i] + B[i + 2]);

A simple image processing pipeline

Overlapped tiling on the iteration space
Comparing with PolyMage

\[
\text{for } (i=1; \ i<N; \ i++) \\
A[i]=f(i); \\
\text{for } (i=2; \ i<N-1; \ i++) \\
\text{for } (i=4; \ i<N-3; \ i++) \\
C[i]=0.25*(B[i-2]+2*B[i]+B[i+2]);
\]

A simple image processing pipeline

Overlapped tiling on the iteration space
Comparing with PolyMage

```c
for (i=1; i<N; i++)
    A[i]=f(i);
for (i=2; i<N-1; i++)
for (i=4; i<N-3; i++)
    C[i]=0.25*(B[i-2]+2*B[i]+B[i+2]);
```

A simple image processing pipeline

Overlapped tiling on the iteration space

We mitigate redundant computation by shrinking the shadows; better performance than a rescheduling-based technique.
### Comparing with PolyMage

<table>
<thead>
<tr>
<th></th>
<th>PolyMage</th>
<th>Our work</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Redundancy</strong></td>
<td>More</td>
<td>Less</td>
</tr>
<tr>
<td><strong>Implementation</strong></td>
<td>With shifting</td>
<td>With/Without shifting</td>
</tr>
<tr>
<td><strong>Applicability</strong></td>
<td>Domain-specific language</td>
<td>General-purpose language</td>
</tr>
<tr>
<td><strong>Targets</strong></td>
<td>CPU</td>
<td>CPU &amp; GPU</td>
</tr>
</tbody>
</table>
Overlapped Tiling With Shifting

Iteration Space

Overlapped tiling with shifting domain

\[
\begin{align*}
S_A (i) & \rightarrow (i) \\
S_B (i) & \rightarrow (i + 1) \\
S_C (i) & \rightarrow (i + 2)
\end{align*}
\]
Overlapped Tiling With Shifting

Iteration Space

Overlapped tiling with shifting
Overlapped Tiling With Shifting

- Iteration Space

- Overlapped tiling with shifting

- Schedule tree

- Polyhedral Implementation of the Overlapped Tiling

- January 26, 2018 7 / 13
Overlapped Tiling With Shifting

Overlapped tiling with shifting

Sequence:
- $S_A(i) \rightarrow (i/\beta b)$
- $S_B(i) \rightarrow ((i + 1)/\beta b)$
- $S_C(i) \rightarrow ((i + 2)/\beta b)$

Schedule Tree:
- $S_A(i)$
  - expansion: $[S_A(i) \rightarrow S_A(j)]$
- $S_B(i)$
  - expansion: $[S_B(i) \rightarrow S_B(j)]$
- $S_C(i)$
  - $[S_C(i) \rightarrow (i + 2)]$
Overlapped Tiling Without Shifting

Iteration space
Overlapped Tiling Without Shifting

Iteration space

Overlapped tiling without shifting
Overlapped Tiling Without Shifting

Iteration space

Overlapped tiling without shifting

domain

sequence

expansion: \([S_A(i) \rightarrow (i/bb); S_B(i) \rightarrow (i/bb); S_C(i) \rightarrow (i/bb)]\)

\(S_A(i)\)

expansion: \([S_A(i) \rightarrow S_A(j)]\)

\(S_B(i)\)

expansion: \([S_B(i) \rightarrow S_B(j)]\)

\(S_C(i)\)

\([S_C(i) \rightarrow (i)]\)

Schedule Tree
Overlapped Tiling Without Shifting

Iteration space

Overlapped tiling without shifting

Domain

Sequence

Schedule Tree

expansion: $[S_A(i) \rightarrow S_A(j)]$

expansion: $[S_B(i) \rightarrow S_B(j)]$

$[S_C(i) \rightarrow (i)]$
Experimental Setup and Methodology

- Code generator: PPCG (version ppcg-0.07-26-g236d559).
- Architecture:
  - CPU: 32-core Intel Xeon(R) E5-2683 v4 @2.10GHz
  - GPU: NVIDIA Quadro K4000
- Compilation:
  - CPU: ICC18.0 (-O3 -xHost -qopenmp -ipo)
  - GPU: NVCC9.0 (-O3)
- Baseline:
  - sequential PolyMage naive code (without tiling) [MVB15]
  - CUDA code generated by PPCG (parallelogram tiling) [VCJC⁺13]
- Comparison:
  - Halide manual and automatic scheduling [RKBA⁺13], PolyMage naive and optimized scheduling [MVB15]
Performance Comparison on CPU

Performance Comparison of Unsharp Mask

Speedup

Halide manual  Halide automatic  PolyMage naive  PolyMage opt  Our work

Experimental Results
Please refer to our poster for more experimental data on the remaining benchmarks, with a detailed comparison with the state of the art.
Performance Comparison of All Benchmarks on GPU

Our technique is also applicable to iterated stencils. Please check on our poster for the evaluation on both CPU and GPU.
We implemented a general purpose overlapped tiling technique in a polyhedral framework.

We implemented overlapped tiling technique with/without shifting.

Our work can generate codes for both CPU and GPU.

We get better performance due to less redundant computation.

Future work: finish the experiments and prepare the paper submission.
References


