Automated GPU Kernel Transformations in Large-Scale Production Stencil Applications

Mohamed Wahib, PI: Naoya Maruyama
RIKEN Advanced Institute for Computational Science
Kobe, Japan

5th April 2016
HPC Applications

Performance Limitations, Patterns, Requirements, …, etc

Applications → GAP → Accelerators

Performance Models, DSL(s), Tools, Abstractions, …, etc
Observation >> Challenges >> Solution

- **Where**: A class of HPC stencil-based applications
  - Regular stencils on fixed grids (memory-bound)
    - Ex: climate models, Seismic simulations, 3D-FDTD
  - Narrow yet important class of applications

- **What**: Series of tight loops
  - Data arrays are reused in those series of loops

- **Relevance**: Potential for reducing redundancy
  - Up to 40% in some applications
Observation >> Challenges >> Solution

- Existing methods for loop fusion
  - Small scale (affine loops)
  - Optimized in isolation

- Transform code to exploit chance for data reuse
  - Dozens of loops and data arrays
  - Constraints from applications
    - Ex: Data dependency and order-of-execution
  - Constraints from architecture
    - Ex: No. of registers and capacity of on-chip memory
  - Automation: to be a practical choice
Framework for automated kernel transformations
  ▪ Applied on GPU kernels
  ▪ Optimize for inter-kernel data reuse
  ▪ Scalable
    ▪ In terms of number of kernels and data arrays

Key results: Six production stencil applications
  ▪ Speedups ranging between 1.12x and 1.76x
  ▪ Enabled by automation
Motivation

Motivation for using different kernels and arrays for various applications:

<table>
<thead>
<tr>
<th>App.</th>
<th>No. Kernels</th>
<th>No. Arrays</th>
<th>Redundancy</th>
</tr>
</thead>
<tbody>
<tr>
<td>SCALE</td>
<td>142</td>
<td>64</td>
<td>41%</td>
</tr>
<tr>
<td>WRF</td>
<td>122</td>
<td>46</td>
<td>24%</td>
</tr>
<tr>
<td>ASUCA</td>
<td>115</td>
<td>58</td>
<td>17%</td>
</tr>
<tr>
<td>MITgcm</td>
<td>94</td>
<td>31</td>
<td>22%</td>
</tr>
<tr>
<td>HOMME</td>
<td>43</td>
<td>27</td>
<td>21%</td>
</tr>
<tr>
<td>COSMO</td>
<td>35</td>
<td>24</td>
<td>38%</td>
</tr>
</tbody>
</table>

Up to 100s

Tens

1.2x~1.67
Overview

Input code → Expose Hidden Locality → Find Best Transformation → Exploit Hidden Locality → Output code

via

Code Analysis → via → Performance Projection → via → Code Transformation
Types of transformation

Kernel Fusion
- Kernel 1
  - Data 1
- Kernel 2
  - Data 2
- Kernel 1+2
  - Data
Data held in on-chip memory

Kernel Fission
- Kernel 1+2
- Kernel 1
  - Data 1
- Kernel 2
  - Data 2
Data held in off-chip memory

M. Wahib and N. Maruyama, Scalable Kernel Fusion for Memory-Bound GPU Applications, SC’14
Transformation (2 of 3)

Kernel A

```c
__global__ Kern_A(R, T, Q, P, V, W, coff, nz, ox, oy) {
    int i = blockIdx.x*blockDim.x+threadIdx.x+ox;
    int j = blockIdx.y*blockDim.y+threadIdx.y+oy;
    for(int k=0; k<nz;k++) {
        R[i][j][k] = coff*(T[i-1][j][k] + T[i][j-1][k])
            + T[i][j-1][k]);
        P[i][j][k] = (Q[i-1][j][k] * Q[i][j-1][k]
            / Q[i][j][k])) + (Q[i][j][k]
            / Q[i-1][j][k] * Q[i][j-1][k])
        W[i][j][k] = min(V[i-1][j][k], V[i][j-1][k],
            V[i][j][k]);
    }
}
```

Kernel B

```c
__global__ Kern_B(T, Q, V, U, coff, nz,ox,oy) {
    int i = blockIdx.x*blockDim.x+threadIdx.x+ox;
    int j = blockIdx.y*blockDim.y+threadIdx.y+oy;
    for(int k=0; k<nz;k++) {
        U[i][j][k] = coff * (T[i-1][j][k] + T[i][j][k]
            + T[i][j-1][k])
            \ - (Q[i][j][k] * (Q[i-1][j][k] - Q[i][j-1][k]))
            *(V[i-1][j][k] * V[i][j-1][k] / V[i][j][k]);
    }
}
```

Kernel X

```c
__global__ Kern_X(R, T, Q, P, V, U, W, coff, nz, ox, oy) {
    int bx = blockDim.x, by = blockDim.y;
    __shared__ double s_T[bx,by], s_Q[bx,by];
    __shared__ double s_V[bx,by];
    double xT, yT, xQ, yQ, xV, yV;
    int tx = threadIdx.x, ty = threadIdx.y;
    int i = blockIdx.x*blockDim.x + tx + ox;
    int j = blockIdx.y*blockDim.y + ty + oy;
    for(int k=0; k<nz;k++) {
        if(tx == 0) {
            xT=T[i-1][j][k]; xQ =Q[i-1][j][k];
            xV=V[i-1][j][k];
        }
        double s_T = 0; s_Q = 0; s_V = 0;
        s_T[tx][ty] = T[i][j][k];
        s_Q[tx][ty] = Q[i][j][k];
        s_V[tx][ty] = V[i][j][k];
        __syncthreads();
        R[i][j][k] = coff*(xT + s_T[tx][ty] + yT);
        P[i][j][k] = (xQ * yQ / s_Q[tx][ty])
            + (s_Q[tx][ty] / xQ * yQ);
    }
}
```
Transformation (3 of 3)

Before Fission

```c
__global__ Kern_A(R, S, T, Q, P, V, U, W, nz, c){
    int i = blockIdx.x*blockDim.x + threadIdx.x;
    int j = blockIdx.y*blockDim.y + threadIdx.y;
    for(int k=0; k<nz;k++) {
        R[i][j][k] = S[i][j][k] * (V[i-1][j][k]/V[i][j][k]);
        W[i][j][k] = V[i-1][j][k] - V[i][j][k];
        P[i][j][k] = c*(Q[i-1][j][k] + Q[i][j-1][k] + Q[i][j][k] + Q[i+1][j][k] + Q[i][j+1][k]);
        U[i][j][k] = T[i][j][k] * Q[i][j][k] * (Q[i-1][j][k] - Q[i][j-1][k]);
    }
}
```

After Fission

```c
__global__ Kern_X(R, S, V, W, nz){
    int i= //absolute index in X-Dir
    int j= //absolute index in y-Dir
    for(int k=0; k<nz;k++) {
        R[i][j][k] = S[i][j][k] * (V[i-1][j][k] / V[i][j][k]);
        W[i][j][k] = V[i-1][j][k] - V[i][j][k];
    }
}
```

```c
__global__ Kern_Y(P, Q, U, T, nz, c){
    int i= //absolute index in X-Dir
    int j= //absolute index in y-Dir
    for(int k=0; k<nz;k++) {
        P[i][j][k] = c*( Q[i-1][j][k] + Q[i][j-1][k]
            + Q[i][j][k] + Q[i+1][j][k] + Q[i][j+1][k]);
        U[i][j][k] = T[i][j][k] - Q[i][j][k]
            * (Q[i-1][j][k] - Q[i][j-1][k]);
    }
}
```
Why Kernel Fission?
Why Kernel Fission?

- Kernel Fission
  - Useful for kernels hitting memory limit
  - Lazy fission

Feasible Solutions

Solution Space

Apply Fission

M. Wahib and N. Maruyama, Automated GPU Kernel Transformations in Large-Scale Production Stencil Applications, HPDC’15
Why kernels not already fused?

Why are kernels not "originally" fused?

“It is interesting that the way the programmer collectively designs his application is NOT what is best for performance. What is more interesting is that he/she doesn’t know.”

---

M. Wahib and N. Maruyama, Scalable Kernel Fusion for Memory-Bound GPU Applications, SC’14
Kernel Fusion as an Optimization Problem

- **Kernel Fusion**: Combinatorial Optimization Problem
  - **Map**: original kernels $\rightarrow$ new kernels
  - **Input**: Original kernels + Data arrays
  - **Output**: New kernels
  - **Objective**: Minimize total runtime of new kernels
  - **Constraints**: Problem-related + Architecture-related
Challenges: Architecture-Specific

- **Ex 1: Modeling performance of new kernels**
  - Lightweight
  - Use existing knowledge
  - Generate new kernels then estimate?

- **Ex 2: Thread blocks and shared memory**
  
  i- Allocate D in Shared Memory
  ii- Compute $D = X(A,B)$
  iii- Compute $E = Y(C,D)$

i- Allocate $D'$ in Shared Memory  
($D' = D + \text{halo layer}$)
ii- Compute $D' = X(A,B)$
iii- Compute $E = Y(C,D)$
Performance-Bound Projection Model

- Performance models (cycle-accurate) are not lightweight
  - Projecting performance for each individual in all generations is prohibitive

- A lightweight codeless performance bound:
  - No form of code representation for fused kernels
  - Roofline? A simple model? … NOT enough
  - Populate solution space with false positives

- Extend existing model* to memory-bound stencil kernels
  - Projects a bound on achievable performance

* J. Lai et al. Performance Upper Bound Analysis and Optimization of SGEMM on Kepler GPU, CGO'16

Fused Kernels → Abstraction → Performance Model

Projecting achievable performance bound for stencil kernels over GPU

Fused Kernels → Performance + architecture Data → Project performance bound
End-to-End Transformation (2 of 2)

- **Framework main features**
  - Fully automated
  - Components can be replaced (Ex: GGA)
  - Can easily extend OpenACC
    - OpenACC is “Descriptive”

- **Programmer Intervention**
  - Different applications >> Different features
  - Amend output of Step $x$ → input to Step $x+1$
Implementation

ROSE Compiler @ LLNL: [http://www.rosecompiler.org](http://www.rosecompiler.org)
# Results

- Production applications from different domains

<table>
<thead>
<tr>
<th>App.</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>SCALE</strong> [Weather]</td>
<td>Next generation mesoscale weather model [Four years in development]</td>
</tr>
<tr>
<td><strong>HOMME</strong> [Climate]</td>
<td>Dynamical core of Community Atmospheric Model (CAM)</td>
</tr>
<tr>
<td><strong>Fluam</strong> [Hydrodynamics]</td>
<td>A fluctuating particle hydrodynamics application based on an hybrid Eulerian-Lagrangian approach</td>
</tr>
<tr>
<td><strong>MITgcm</strong> [Oceanic]</td>
<td>An oceanic general circulation model relying on a finite volume numerical method [18 years in development]</td>
</tr>
<tr>
<td><strong>AWP-ODC-GPU</strong> [Seismic]</td>
<td>An earthquake wave propagation simulator [ACM Gordon Bell finalist]</td>
</tr>
<tr>
<td><strong>B-CALM</strong> [FDTD]</td>
<td>A 3D-FDTD simulator which models the permittivity of dispersive material</td>
</tr>
</tbody>
</table>
## Results (Cont.)

- **Production applications from different domains**

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>SCALE</td>
<td>103</td>
<td>142</td>
<td>63</td>
<td>0.07</td>
</tr>
<tr>
<td>HOMME</td>
<td>51</td>
<td>43</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Fluam</td>
<td>201</td>
<td>169</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MITgcm</td>
<td>61</td>
<td>379</td>
<td></td>
<td></td>
</tr>
<tr>
<td>AWP-ODC-GPU</td>
<td>48</td>
<td>12</td>
<td>24</td>
<td>1.062</td>
</tr>
<tr>
<td>B-CALM</td>
<td>58</td>
<td>23</td>
<td>24</td>
<td>0.783</td>
</tr>
</tbody>
</table>

**Significant Fission**
Nvidia K40 speedup compared to baseline CUDA version with no kernel fission/fusion
K40 speedup for automated vs. manual kernel filtering compared to baseline CUDA version with no kernel fission/fusion
## Results (Cont.)

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>SCALE</td>
<td>38</td>
<td>14</td>
<td>0.65</td>
<td>0.80</td>
</tr>
<tr>
<td>HOMME</td>
<td>9</td>
<td>4</td>
<td>0.55</td>
<td>0.85</td>
</tr>
<tr>
<td>Fluam</td>
<td>17</td>
<td>11</td>
<td>0.81</td>
<td>0.90</td>
</tr>
<tr>
<td>MITgcm</td>
<td>6</td>
<td>3</td>
<td>0.95</td>
<td>0.96</td>
</tr>
<tr>
<td>AWP-ODC-GPU</td>
<td>3</td>
<td>2</td>
<td>0.75</td>
<td>0.77</td>
</tr>
<tr>
<td>B-CALM</td>
<td>3</td>
<td>0</td>
<td>0.72</td>
<td>0.72</td>
</tr>
</tbody>
</table>

Occupyance range = [0,1] (Higher is better)
Summary

- Framework for **automated** kernel transformations
  - Applied on GPU kernels
  - Optimize for inter-kernel data reuse
  - Six production applications (1.12x and 1.76x)
Limitations

- Pointer aliasing
  - Restrict
  - Solution: inspired by Stream applications

- Supported stencils
  - Dense multidimensional Cartesian grids (unit-stride)

- Sensitivity to input
  - Should be fine for typical stencils (weak scaling)
  - Possible sensitivity to specific input values
Future Work

- Improve the framework
  - Address limitations
  - Improve code generation
  - Other accelerators

- Follow us on [www.github.com/wahibium/kff](http://www.github.com/wahibium/kff)
  - Experimental version
Thank you for listening.

Questions?