Taking on the dwarfs: Advocating domain-specific frameworks for many-core HPC

Graham Pullan
Tobias Brandvik
Department of Engineering
Overview

• Dwarfs
• Structured grid stencil operations
• Templating
Overview

- Dwarfs
- Structured grid stencil operations
- Templating

All with CFD as the target application
The mind of a domain scientist

The Task

The Code

HPC hardware / software
The mind of a domain scientist

e.g. for the present speaker:

- Jet-engine aerodynamics
- Turbomachinery CFD
- Fortran / MPI / GPU / CUDA
The mind of a domain scientist

e.g. for Steven Gratton:

- The origins of the Universe
- Analysis of satellite data
- C / MPI / GPU / CUDA

Importance
How to best share our experiences?

• Each of us is motivated by a step-change in task performance (results per $, results per W)

• But, presented by task, overlap of techniques is not obvious

• Dwarfs provide the necessary taxonomy
Dwarfs – Disney (1937)
Dwarfs – Colella (2004)

- Structured
- N-Body
- Spectral
- Unstructured
- Monte-Carlo
- Sparse LA
- Dense LA
Turbostream CFD code

- Finite volume structured grid code
- Relevant dwarfs:
  - In the bulk – structured grid
  - At the boundary – sparse linear algebra
Finite volume CFD

Divide the volume into cells

Blade

Flow

Divide the volume into cells
Governing equations for each cell
Governing equations for each cell

Conserve:
- Mass
- Momentum
- Energy
Example: mass conservation

- Evaluate mass fluxes on each face

\[ F_{\text{mass}} = \frac{A}{4} \sum \rho V_n \]
Example: mass conservation

- Sum fluxes on faces to find density change in cell

\[ \Delta \rho_{\text{cell}} = \Delta t \sum F_{\text{mass}} \]
Example: mass conservation

- Update density

\[ \Delta \rho_{\text{node}} = \frac{1}{8} \sum \Delta \rho_{\text{cell}} \]
Similarity of steps

Each step uses data from surrounding nodes – “stencil” operation
Structured grid strategy

- Divide up domain
  - each sub-domain to a thread block
  - update nodes in sub-domain with most efficient stencil operation we can come up with
    (make effective use of shared mem)
CUDA strategy (after Williams et al, 2007)

- For each block, start a plane of threads (an i-k plane)
- Load three planes into shared memory
  - Compute one plane
  - Load next plane into shared memory (swap out first plane)
    - Compute next plane
- Repeat, moving along j direction
CUDA strategy

1. Load 2 rows into shared mem

Zone solved by Block 1
CUDA strategy

1. Load 2 rows into shared mem

2. To compute points, load next row into shared mem

Zone solved by Block 1
CUDA strategy

1. Load 2 rows into shared mem

2. To compute points, load next row into shared mem

3. Move up domain, row by row (load new row into shared mem, drop lowest row out of shared mem)
CUDA strategy

1. Load 2 rows into shared mem

2. To compute points, load next row into shared mem

3. Move up domain, row by row (load new row into shared mem, drop lowest row out of shared mem)
CUDA strategy

1. Load 2 rows into shared mem

2. To compute red points, load next row into shared mem

3. Move up domain, row by row (load new row into shared mem, drop lowest row out of shared mem)
CUDA strategy

1. Load 2 rows into shared mem

2. To compute red points, load next row into shared mem

3. Move up domain, row by row (load new row into shared mem, drop lowest row out of shared mem)
__global__ void stencil_kernel(float sf, float *a_data, float *b_data) {
    __shared__ float a[16][3][5];
    i = (int) threadIdx.x;
    k = (int) threadIdx.y;
    a[i][0][k] = a_data[i0m10];
    a[i][1][k] = a_data[i000];
    /* begin loop in j-direction */
    a[i][2][k] = a_data[i0p10];
    __syncthreads();
    /* compute */
    b_data[i000] =
        sf1*a[i][1][k] + sfd6*(a[im1][1][k] + a[ip1][1][k] + a[i][0][k] +
        a[i][2][k] + a[i][1][km1] + a[i][1][kp1])
    /* repeat: load j-plane, syncthreads, compute...*/
}
CUDA code (nearest neighbour stencil)

```c
__global__ void stencil_kernel(float sf, float *a_data, float *b_data)

__shared__ float a[16][3][5];

i = (int) threadIdx.x;
k = (int) threadIdx.y;
a[i][0][k] = a_data[i0m10];
a[i][1][k] = a_data[i000];
/* begin loop in j-direction */
a[i][2][k] = a_data[i0p10];
__syncthreads();
/* compute */
b_data[i000] =
    sf1*a[i][1][k] + sfd6*(a[im1][1][k] +
    a[ip1][1][k] + a[i][0][k] +
    a[i][2][k] + a[i][1][km1] + a[i][1][kp1])
/* repeat: load j-plane, syncthreads, compute...*/
```
CUDA code (nearest neighbour stencil)

```c
__global__ void stencil_kernel(float sf, float *a_data, float *b_data) {
    __shared__ float a[16][3][5];

    i = (int) threadIdx.x;
    k = (int) threadIdx.y;

    a[i][0][k] = a_data[i0m10];
    a[i][1][k] = a_data[i000];
    /* begin loop in j-direction */
    a[i][2][k] = a_data[i0p10];
    __syncthreads();
    /* compute */
    b_data[i000] =
        sf1*a[i][1][k] + sfd6*(a[im1][1][k] +
        a[ip1][1][k] + a[i][0][k] +
        a[i][2][k] + a[i][1][km1] + a[i][1][kp1])
    /* repeat: load j-plane, syncthreads, compute...*/
```

get i,k thread indices
CUDA code (nearest neighbour stencil)

```c
__global__ void stencil_kernel(float sf, float *a_data, float *b_data){
  __shared__ float a[16][3][5];
  i = (int) threadIdx.x;
  k = (int) threadIdx.y;

  a[i][0][k] = a_data[i0m10];
  a[i][1][k] = a_data[i000];

  /* load initial 2 planes */

  /* begin loop in j-direction */
  a[i][2][k] = a_data[i0p10];

  __syncthreads();

  /* compute */
  b_data[i000] =
    sf1*a[i][1][k] + sfd6*(a[im1][1][k] +
        a[ip1][1][k] + a[i][0][k] +
        a[i][2][k] + a[i][1][km1] + a[i][1][kp1])

  /* repeat: load j-plane, syncthreads, compute...*/
```
CUDA code (nearest neighbour stencil)

```c
__global__ void stencil_kernel(float sf, float *a_data, float *b_data)
{
    __shared__ float a[16][3][5];
    i = (int) threadIdx.x;
    k = (int) threadIdx.y;
    a[i][0][k] = a_data[i0m10];
    a[i][1][k] = a_data[i000];
    /* begin loop in j-direction */
    a[i][2][k] = a_data[i0p10];
    __syncthreads();
    /* compute */
    b_data[i000] =
        sf1*a[i][1][k] + sfd6*(a[im1][1][k] + a[i][0][k] + a[i][1][km1] + a[i][1][kp1])
    /* repeat: load j-plane, syncthreads, compute...*/
}
```

**main loop:**

- load next plane
- syncthreads (whole plane loaded)

**compute result (if not a halo node)**

/* repeat: load j-plane, syncthreads, compute...*/
Motivation for “SBLOCK” framework

- CFD code will have many stencil kernels
- All look (almost) the same
- During development – several optimization strategies might be tried
- We want to decouple the stencil task from the hardware target
Source-to-source compilation

- The stencil definition is transformed at compile-time into code that can run on the chosen processor
- The transformation is performed by filling in a pre-defined template using the stencil definition
Source-to-source compilation

- The stencil definition is transformed at compile-time into code that can run on the chosen processor.
- The transformation is performed by filling in a pre-defined template using the stencil definition.

Stencil definition → CPU template → CPU source (.c) → X source
Stencil definition → GPU template → GPU source (.cu)
Stencil definition → X template → X source
A Python-based template engine – “Cheetah”

fortran_tmpl.tmpl:

```fortran
WRITE(6,*) 'message'
STOP
END
```

make python module:

```
cheetah compile fortran_tmpl
```
A Python-based template engine – “Cheetah”

fortran_tmpl.tmpl:

```fortran
WRITE(6,*) '$message'
STOP
END
```

html_tmpl.tmpl:

```html
<HTML>
<HEAD><TITLE>Test</TITLE></HEAD>
<BODY>
<p>$message</p>
</BODY>
</HTML>
```

make python module:

```
cheetah compile fortran_tmpl
```

make python module:

```
cheetah compile html_tmpl
```
A Python-based template engine – “Cheetah”

make_fortran.py:
```
from fortran_tmpl import *
t=fortran_template()
t.message="Hello"
print t
```

make_html.py:
```
from html_tmpl import *
t=html_template()
t.message="Hello"
print t
```
A Python-based template engine – “Cheetah”

```python
from fortran_tmpl import *
t=fortran_template()
t.message="Hello"
print t
```

**python** make_fortran.py gives

```
WRITE(6,*) 'Hello'
STOP
END
```

```python
from html_tmpl import *
t=html_template()
t.message="Hello"
print t
```

**python** make_html.py gives

```
<HTML>
<HEAD><TITLE>Test</TITLE></HEAD>
<BODY>
<p>Hello</p>
</BODY>
</HTML>
```
Example SBLOCK stencil definition

```python
kind = "stencil"
bpin = ["a"]
bpout = ["b"]
lookup = ((1,0, 0), (0, 0, 0), (1,0, 0), (0, 1,0),
          (0, 1, 0), (0, 0, 1), (0, 0, 1))

calc = {"lvalue": "b",
        "rvalue": """"sf1*a[0][0][0] +
sfd6*(a[1][0][0] + a[1][0][0] +
a[0][1][0] + a[0][1][0] +
a[0][0][1] + a[0][0][1])""""}
Turbostream

- 3000 lines of stencil definitions (~15 different stencil kernels)
- Code generated from stencil definitions is 15,000 lines
- Additional 5000 lines of C for boundary conditions, file I/O etc.
- Source code is very similar to TBLOCK – every subroutine has an equivalent stencil definition
Single-processor performance

<table>
<thead>
<tr>
<th>Solver</th>
<th>Processor</th>
<th>Time/node/step</th>
</tr>
</thead>
<tbody>
<tr>
<td>TBLOCK</td>
<td>Intel Xeon 2.33 GHz</td>
<td>$5.1 \times 10^{-7}$ s</td>
</tr>
<tr>
<td>Turbostream</td>
<td>NVIDIA GT200</td>
<td>$2.7 \times 10^{-8}$ s</td>
</tr>
</tbody>
</table>

- TBLOCK uses all four cores on the CPU through MPI
- Turbostream is ~20 times faster
Multi-processor performance

- Benchmark case is an unsteady simulation of a turbine stage
Multi-processor performance

- 16 NVIDIA G200 GPUs, 1 Gb/s Ethernet
- Weak scaling: 6 million grid nodes per GPU
- Strong scaling: 6 million grid nodes in total
Desktop run times

Steady model

3 x GT200: 7 minutes
2 x Xeon quad: 210 minutes

Unsteady model

3 x GT200: 1 hour
2 x Xeon quad: 30 hours
Summary

• Dwarfs: a taxonomy for sharing experiences / techniques between practitioners from different fields
Summary

- Dwarfs: a taxonomy for sharing experiences / techniques between practitioners from different fields
- Structured grid dwarf: Plane-by-plane (cyclic queue) approach yields good results (Datta et al SC08)
Summary

- Dwarfs: a taxonomy for sharing experiences / techniques between practitioners from different fields

- Structured grid dwarf: Plane-by-plane (cyclic queue) approach yields good results (Datta et al SC08)

- Templating can:
  - Save time during development
  - Makes porting to different languages / platforms painless
Summary

- Dwarfs: a taxonomy for sharing experiences / techniques between practitioners from different fields

- Structured grid dwarf: Plane-by-plane (cyclic queue) approach yields good results (Datta et al SC08)

- Templating can:
  - Save time during development
  - Makes porting to different languages / platforms painless

- Resulting CFD code shows 20x speedup (GT200 vs 2.33GHz Intel quad) compared to legacy Fortran-MPI code
Finally

2nd UK CUDA Developers’ Conference

December 2010, Cambridge