Vectorizing Unstructured Mesh Computations for Many-core Architectures

István Z. Reguly, E. László, G. R. Mudalige, M. B. Giles
University of Oxford
Oxford e-Research Centre
Motivation

• Huge variety of hardware

• Old & New programming languages and abstractions

  • Do old ones have what it takes to be efficient on new hardware?

  • Are new ones getting accepted?

• Disparity between the way of programming and what the hardware can do
Bridging the gap

- Plethora of parallel programming languages, abstractions and execution models
  - Confusing boundaries, one language is usually only well-suited for one target architecture
- Are compilers supposed to bridge the gap?
  - On simple, regular problems they do an adequate job
  - On complex, irregular ones not so much…
Unstructured mesh computations

From the viewpoint of a “unit of work” - kernel

- Direct & Indirect
- Gather & Scatter
Unstructured mesh computations

- Irregular accesses
- Connectivity only known at run-time
- We want to parallelize execution
  - Multi-level parallelism
  - Deep memory hierarchies
- Data dependencies and race conditions
Execution

Take the following (oversimplified) example

```c
for (int i = 0; i < set_size; i++) {
    double *n1 = &coords[2*cell2vertex[i*4+0]];
    double *n2 = &coords[2*cell2vertex[i*4+1]];
    double *n3 = &coords[2*cell2vertex[i*4+2]];
    double *n4 = &coords[2*cell2vertex[i*4+3]];
    double *ce = &celldata[4*i];
    double edge1[1], edge2[1], edge3[1], edge4[1];
    //inlined user kernel
    double dx1 = n1[0] - n2[0];
    double dy1 = n1[1] - n2[1];
    double dx2 = n3[0] - n4[0];
    double dy2 = n3[1] - n4[1];
    edge1[0] += (dx+dy)*ce[0];
    edge2[0] += (dx-dy)*ce[1];
    edge3[0] += (dx+dy)*ce[2];
    edge4[0] += (dx-dy)*ce[3];
    //apply increments
    deltas[cell2edge[i*4+0]] += edge1[0];
    deltas[cell2edge[i*4+1]] += edge2[0];
    deltas[cell2edge[i*4+2]] += edge3[0];
    deltas[cell2edge[i*4+3]] += edge4[0];
}
```
How to map this to …

- Parallel programming abstractions
  - Distributed memory, coarse-grained shared memory and fine-grained shared memory

- Hardware execution models
  - Cache utilization, coherency, cores, SIMD execution units, communication and synchronization mechanisms
Three levels of parallelism
GPU - what the hardware will do for you

- Using the SIMT parallel programming model, the previous scheme can be easily expressed
  - Data reuse through cache or scratchpad memory
  - Colored updates using synchronisation
  - CUDA or OpenCL, easy to implement, simple, clean code
- Maps quite well to GPU hardware, hardware does all the gather and scatter for you
  - But hardware changes, new optimisations have to be implemented
__global__ void kernel_wrap(...) {
    int i = threadIdx.x + blockIdx.x*blockDim.x
    double *n1 = &coords[2*cell2vertex[i*4+0]];
    double *n2 = &coords[2*cell2vertex[i*4+1]];
    double *n3 = &coords[2*cell2vertex[i*4+2]];
    double *n4 = &coords[2*cell2vertex[i*4+3]];
    double *ce = &celldata[4*i];
    double edge1[1],edge2[1],edge3[1],edge4[1];
    //inlined user kernel
    double dx1 = n1[0] - n2[0];
    double dy1 = n1[1] - n2[1];
    double dx2 = n3[0] - n4[0];
    double dy2 = n3[1] - n4[1];
    edge1[0] += (dx+dy)*ce[0];
    edge2[0] += (dx-dy)*ce[1];
    edge3[0] += (dx+dy)*ce[2];
    edge4[0] += (dx-dy)*ce[3];
    //apply increments
    for (int c = 0; ic < ncolors; c++) {
        __syncthreads();
        if (c == mycolor) {
            deltas[cell2edge[i*4+0]] += edge1[0];
            deltas[cell2edge[i*4+1]] += edge2[0];
            deltas[cell2edge[i*4+2]] += edge3[0];
            deltas[cell2edge[i*4+3]] += edge4[0];
        }
    }
}
CPU - coarse grained

- Using either distributed memory (MPI) or coarse-grained shared memory (OpenMP) is fairly easy.
  - Have do handle data dependencies or data races at a high level as discussed.
  - Each process/thread iterates over an execution set serially, good cache locality.
- Can be handled with generic code.
- Specialized code can enable more compiler optimisations.
CPU and GPU baseline

![Runtime comparison graph]

- Old
- New
- Fermi C2070
- Fermi K40
- Kepler K40

Runtime (seconds)

MPI
OpenMP
CUDA
CPU and GPU baseline

Bandwidth in GB/s and Compute in GFLOPS

<table>
<thead>
<tr>
<th>Kernel</th>
<th>pure MPI</th>
<th>CUDA</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Time</td>
<td>BW</td>
</tr>
<tr>
<td>save_soln</td>
<td>0.99</td>
<td>46.55</td>
</tr>
<tr>
<td>adt_calc</td>
<td>6.3</td>
<td>18.24</td>
</tr>
<tr>
<td>res_calc</td>
<td>6.58</td>
<td>56.72</td>
</tr>
<tr>
<td>bres_calc</td>
<td>0.03</td>
<td>27.15</td>
</tr>
<tr>
<td>update</td>
<td>3.23</td>
<td>60.62</td>
</tr>
</tbody>
</table>
Managing data for vectorization

Vertex data

| xy | xy | xy | xy | xy | xy | xy | xy | xy | xy | xy | xy | xy | xy | xy | xy |

Cell data

| p  | p  | p  | p  |     | [0] |
| q  | q  | q  | q  |     | [1] |
| r  | r  | r  | r  |     | [2] |
| s  | s  | s  | s  |     | [3] |

cell 1 cell 2 cell 3 cell 4

Edge data

| u  | u  | u  | u  | u  |     |

vectorized cell 1-4
Enabling vectorization

We have to spoon-feed the compiler

```c
for (int i = 0; i < set_size; i += 4) {
    double n1[2][4] = {{coords[2*cell2vertex[(i+0)*4+0]+0],
                        coords[2*cell2vertex[(i+1)*4+0]+0],
                        coords[2*cell2vertex[(i+2)*4+0]+0],
                        coords[2*cell2vertex[(i+3)*4+0]+0]},
                        {coords[2*cell2vertex[(i+0)*4+0]+1],
                        coords[2*cell2vertex[(i+1)*4+0]+1],
                        coords[2*cell2vertex[(i+2)*4+0]+1],
                        coords[2*cell2vertex[(i+3)*4+0]+1]}};
...
#pragma simd
for (int j = 0; j < 4; j++){
    //inlined user kernel
    double dx1 = n1[0][j] - n2[0][j];
    double dy1 = n1[1][j] - n2[1][j];
    double dx2 = n3[0][j] - n4[0][j];
    double dy2 = n3[1][j] - n4[1][j];
    edge1[0][j] += (dx+dy)*ce[0][j];
    edge2[0][j] += (dx-dy)*ce[1][j];
    edge3[0][j] += (dx+dy)*ce[2][j];
    edge4[0][j] += (dx-dy)*ce[3][j];
}
//scatter data
deltas[cell2edge[(i+0)*4+0]] += edge1[0][0];
...}
```

And even that doesn't work consistently…
Vector intrinsics

- Clearly, nobody wants to write such code:

\[
\text{adt} = \text{fabs}(u*dy-v*dx) + c*\sqrt{dx*dx+dy*dy};
\]

\[
\text{adt} = \_\text{mm256_add_pd}(\_\text{mm256_max_pd}(\_\text{mm256_sub_pd}(\_\text{mm256_mul_pd}(u,dy),\_\text{mm256_mul_pd}(v,dx)),\_\text{mm256_sub_pd}(\_\text{mm256_mul_pd}(v,dx),\_\text{mm256_mul_pd}(u,dy))),\_\text{mm256_mul_pd}(c,\_\text{mm256_sqrt_pd}(\_\text{mm256_add_pd}(\_\text{mm256_mul_pd}(dx,dx),\_\text{mm256_mul_pd}(dy,dy)))));
\]

- Fortunately we can use C++ classes and operator overloading

```cpp
class F64vec4 {
protected:
  __m256d vec;
public:
  F64vec4() {}
friend F64vec4 operator *(const F64vec4 &a, const F64vec4 &b) { return _mm256_mul_pd(a,b); }
friend F64vec4 sqrt(const F64vec4 &a) { return _mm256_sqrt_pd(a); }
friend F64vec4 min(const F64vec4 &a, const F64vec4 &b) {
  return _mm256_min_pd(a,b); }
}
static inline float min_horizontal(const F32vec8 &a) {
  F32vec8 temp = _mm256_min_ps(a, _mm256_permute_ps(a, 0xee));
  temp = _mm256_min_ps(temp, _mm256_movehdup_ps(temp));
  return _mm_cvtss_f32(_mm_min_ss(_mm256_castps256_ps128(temp), _mm256_extractf128_ps(temp, 1)));
}
```
Vector intrinsics

Using vector datatypes, the code looks simpler:

```c
for (int i =0; i < set_size; i+=4) {
    doublev n1[2] ={doublev(coords+0, &cell2vertex[i*4], 2, 4),
                    doublev(coords+1, &cell2vertex[i*4], 2, 4)};
    ...

    //inlined user kernel
    doublev dx1 = n1[0] - n2[0];
    doublev dy1 = n1[1] - n2[1];
    doublev dx2 = n3[0] - n4[0];
    doublev dy2 = n3[1] - n4[1];
    edge1[0]+= (dx+dy)*ce[0];
    edge2[0]+= (dx-dy)*ce[1];
    edge3[0]+= (dx+dy)*ce[2];
    edge4[0]+= (dx-dy)*ce[3];

    //scatter data
    scatter(deltas,&cell2edge[i*4], 1, 4);
    ...
}
```

Although all the branching has to be replaced with select() instructions that can be overloaded
CPU vectorized performance

![Bar chart showing runtime performance for CPU1 SP, CPU2 SP, CPU1 DP, and CPU2 DP using different parallelization techniques: MPI, MPI vectorized, OpenMP, OpenMP vectorized, and OpenCL.]
compute throughput. Switching to single precision should
demonstrate that these kernels were, to some extent, bound by
bandwidth values with that of Table 4, it is clear that the
precision, and both problem sizes. Comparing double precision
foil benchmark on CPU 1, both single and double preci-
precision. The perfect 80-110% speedup when going from double to single
ble precision), while the vectorized version shows an almost
computational throughput is the same in single and dou-
was also sped up by 30%, providing further ev-
ent and OpenMP have some overhead due to threading
the overhead of MPI communications is almost completely
faster than the MPI+OpenMP versions, because at this scale
mark in single (SP) and double (DP) precision on CPU
Figure 4:
Figure 5: Performance results comparing the
Table 6: Timing and bandwidth breakdowns for the
Table 7: Timing and bandwidth breakdowns for the
Double(Single) precision breakdowns

<table>
<thead>
<tr>
<th>Kernel</th>
<th>720k cells</th>
<th>2.8M cells</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Time</td>
<td>BW</td>
</tr>
<tr>
<td>save_soln</td>
<td>1.01(0.28)</td>
<td>45(82)</td>
</tr>
<tr>
<td>adt_calc</td>
<td>3.3(1.33)</td>
<td>34(44)</td>
</tr>
<tr>
<td>res_calc</td>
<td>5.06(3.5)</td>
<td>73(59)</td>
</tr>
<tr>
<td>update</td>
<td>3.33(1.5)</td>
<td>59(65)</td>
</tr>
</tbody>
</table>
Xeon Phi - MPI+OpenMP

Figure 6: Performance of the Xeon Phi on the 720k cell mesh in double precision using 100 OpenMP threads per process to give a total of 3000 OpenMP threads.

Figure 7: Performance of the Xeon Phi on the 2.8M cell mesh however reveals that the Xeon Phi is actually quite slow, the reasons for which are currently not well understood. Comparing the runtimes of the small and the large problems, vectorized MPI+OpenMP version speeds up by 2.5 in single and 1.95 in double precision beyond 120. Quadrupling the problem size to 2.8M cells shown in Figure 8 for the best combination of MPI+OpenMP in each vectorized version - which is due to the MPI messaging overhead being even higher, 2-2.2 times in single and 1.7-1.82 times in double precision. Unlike in the case of the CPU, the hybrid approach gives a significant performance boost, but the difference is not as significant as in the CPU case. Performance is generally slightly better compared to the CPU, the CPU does not show signs of being sensitive to being fully utilized; while the problem size is four times bigger, it only takes 2.97 times more time in single precision and 3.25 times more time in double precision to finish the execution for the vectorized MPI+OpenMP version. As mentioned in Section 5.3 of the paper, the vectorized version of the OpenMP program is oblivious to any MPI processes, whereas the non-vectorized version, for MPI+OpenMP the difference is even bigger, 2.64 times in single and 2.28 times in double precision.

Tests included pure MPI execution and different combinations of MPI processes and OpenMP threads, setting the OP2 OpenMP parallelization approach, there is no data to reference the performance differences between the settings of the two approaches.

Due to the nature of the code, the OP2 toolchain has to be overridden.

The vectorized version can exploit new features in the Phi, such as gather instructions, vector reduction, etc., and integrate it seamlessly into the OP2 toolchain.

Through this, we can exploit new features in the Phi, such as gather instructions, vector reduction, etc., and integrate it seamlessly into the OP2 toolchain.
Xeon Phi performance

![Bar chart showing runtime for different configurations.](image-url)
and bandwidth values with that of Table 4, it is clear that the perfect 80-110% speedup when going from double to single computational throughput is the same in single and double precision, we only see a 30-40% di

97%, but in double precision it is only 15-37%; this is due hidden, while OpenMP has some overhead due to threading the overhead of MPI communications is almost completely faster than the MPI+OpenMP versions, because at this scale

Figure 4: the OpenCL backend on a single socket of CPU 1, Airfoil benchmarks in double(single) precision using

Table 7 shows per-loop breakdowns of the vectorized Airfoil benchmark:

<table>
<thead>
<tr>
<th>Kernel</th>
<th>720k cells</th>
<th></th>
<th>2.8M cells</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Time</td>
<td>BW</td>
<td>Time</td>
<td>BW</td>
</tr>
<tr>
<td>save_soln</td>
<td>0.58(0.25)</td>
<td>80(90)</td>
<td>2.18(1.18)</td>
<td>84(76)</td>
</tr>
<tr>
<td>adt_calc</td>
<td>2.0(1.16)</td>
<td>57(50)</td>
<td>6.83(3.33)</td>
<td>68(72)</td>
</tr>
<tr>
<td>res_calc</td>
<td>7.52(5.47)</td>
<td>45(38)</td>
<td>24.1(17.4)</td>
<td>61(48)</td>
</tr>
<tr>
<td>update</td>
<td>2.55(1.93)</td>
<td>77(50)</td>
<td>8.77(4.69)</td>
<td>89(83)</td>
</tr>
</tbody>
</table>

For reference, on the CPU:

<table>
<thead>
<tr>
<th>Kernel</th>
<th>720k cells</th>
<th></th>
<th>2.8M cells</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Time</td>
<td>BW</td>
<td>Time</td>
<td>BW</td>
</tr>
<tr>
<td>save_soln</td>
<td>1.01(0.28)</td>
<td>45(82)</td>
<td>4.1(2.0)</td>
<td>45(45)</td>
</tr>
<tr>
<td>adt_calc</td>
<td>3.3(1.33)</td>
<td>34(44)</td>
<td>12.7(5.2)</td>
<td>37(46)</td>
</tr>
<tr>
<td>res_calc</td>
<td>5.06(3.5)</td>
<td>73(59)</td>
<td>19.5(13.5)</td>
<td>76(62)</td>
</tr>
<tr>
<td>update</td>
<td>3.33(1.5)</td>
<td>59(65)</td>
<td>14.6(7.0)</td>
<td>54(56)</td>
</tr>
</tbody>
</table>
Performance summary
Performance summary

Relative speedup over CPU 1 in double precision

<table>
<thead>
<tr>
<th>Kernel</th>
<th>CPU 1</th>
<th>CPU 2</th>
<th>Xeon Phi</th>
<th>K40</th>
</tr>
</thead>
<tbody>
<tr>
<td>save_soln</td>
<td>1.0</td>
<td>1.04</td>
<td>1.88</td>
<td>5.11</td>
</tr>
<tr>
<td>adt_calc</td>
<td>1.0</td>
<td>1.43</td>
<td>1.87</td>
<td>4.67</td>
</tr>
<tr>
<td>res_calc</td>
<td>1.0</td>
<td>1.33</td>
<td>0.81</td>
<td>1.79</td>
</tr>
<tr>
<td>update</td>
<td>1.0</td>
<td>1.03</td>
<td>1.67</td>
<td>4.49</td>
</tr>
</tbody>
</table>
Code generation

- This is all very nice, but is it generic?
- OP2 abstraction for unstructured grid computations

<table>
<thead>
<tr>
<th>Argument</th>
<th>Type</th>
<th>Indirection property</th>
<th>Direct/indirect argument</th>
<th>Dataset dimensionality</th>
<th>Access type</th>
<th>Argument type</th>
</tr>
</thead>
<tbody>
<tr>
<td>op_par_loop(adt_calc,&quot;adt_calc&quot;,cells,</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>op_arg_dat(p_x, 0,pcell, 2,&quot;double&quot;,OP_READ ),</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>op_arg_dat(p_x, 1,pcell, 2,&quot;double&quot;,OP_READ ),</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>op_arg_dat(p_x, 2,pcell, 2,&quot;double&quot;,OP_READ ),</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>op_arg_dat(p_x, 3,pcell, 2,&quot;double&quot;,OP_READ ),</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>op_arg_dat(p_q, -1,OP_ID, 4,&quot;double&quot;,OP_READ ),</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>op_arg_dat(p_res, 0,ecell, 1,&quot;double&quot;,OP_INC),</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>op_arg_dat(p_res, 1,ecell, 1,&quot;double&quot;,OP_INC),</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>op_arg_dat(p_res, 2,ecell, 1,&quot;double&quot;,OP_INC),</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>op_arg_dat(p_res, 3,ecell, 1,&quot;double&quot;,OP_INC));</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Summary

• Based on high-level specifications and domain specific knowledge, it is possible to automate parallel execution

  • Map to different parallel programming languages, abstractions and execution models using code generation

  • Vectorization for unstructured mesh computations is thus achievable, although far from ideal

    • OpenCL is “nice” but slow - compiler is unable to bridge the gap

    • AVX is “ugly” but fast - we do it instead of the compiler

• Constraints of the hardware are still important, especially the penalty due to serialization when incrementing indirect data

Thank you! Questions?