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 traditional methods 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…
- Domain Specific Languages
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
CPU - coarse grained

- Using either distributed memory (MPI) or coarse-grained shared memory (OpenMP) is fairly easy
  - Have to 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
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
Mapping to SIMT

```c
__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; c < 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 and GPU baseline

Non-linear 2D inviscid airfoil code - 720k cells, 1000 iterations

Dual socket Sandy Bridge Xeon E5-2640
188 GFLOPS
65 GB/s

Tesla K40
2880 CUDA cores
1420 GFLOPS
229 GB/s
Gather & Scatter

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>

Direct

Gather & Scatter

Dual socket Sandy Bridge Xeon E5-2640
188 GFLOPS
65 GB/s

Tesla K40
2880 CUDA cores
1420 GFLOPS
229 GB/s
Code generation

• This is all very nice, but is it generic?

• OP2 abstraction for unstructured grid computations

```
Function name
op_par_loop(adt_calc,"adt_calc",cells,
    op_arg_dat(p_x, 0,pcell, 2,"double",OP_READ ),
    op_arg_dat(p_x, 1,pcell, 2,"double",OP_READ ),
    op_arg_dat(p_x, 2,pcell, 2,"double",OP_READ ),
    op_arg_dat(p_x, 3,pcell, 2,"double",OP_READ ),
    op_arg_dat(p_q, -1,OP_ID, 4,"double",OP_READ ),
    op_arg_dat(p_res, 0,ecell, 1,"double",OP_INC),
    op_arg_dat(p_res, 1,ecell, 1,"double",OP_INC),
    op_arg_dat(p_res, 2,ecell, 1,"double",OP_INC),
    op_arg_dat(p_res, 3,ecell, 1,"double",OP_INC));
```
```c
void op_par_loop_adt_calc(char const *name, op_set set,
                         op_arg arg0, op_arg arg1, op_arg arg2,
                         op_arg arg3, op_arg arg4, op_arg arg5){

    int nargs = 6;
    op_arg args[6] = {arg0, arg1, arg2, arg3, arg4, arg5};

    // initialise timers
    double cpu_t1, cpu_t2, wall_t1, wall_t2;
    op_timing_realloc(1);
    op_timers_core(&cpu_t1, &wall_t1);

    int exec_size = op_mpi_halo_exchanges(set, nargs, args);

    for (int n = 0; n < exec_size; n++) {
        if (n == set->core_size) {
            op_mpi_wait_all(nargs, args);
        }
        int map0idx = arg0.map_data[n * arg0.map->dim + 0];
        int map1idx = arg0.map_data[n * arg0.map->dim + 1];
        int map2idx = arg0.map_data[n * arg0.map->dim + 2];
        int map3idx = arg0.map_data[n * arg0.map->dim + 3];

        adt_calc(
            &((double*)arg0.data)[2 * map0idx],
            &((double*)arg0.data)[2 * map1idx],
            &((double*)arg0.data)[2 * map2idx],
            &((double*)arg0.data)[2 * map3idx],
            &((double*)arg4.data)[4 * n],
            &((double*)arg5.data)[1 * n]);
    }

    // update kernel record
    op_timers_core(&cpu_t2, &wall_t2);
    OP_kernels[1].name = name;
    OP_kernels[1].count += 1;
    OP_kernels[1].time += wall_t2 - wall_t1;
}
```

Only depends on #of args

Setting up indices for indirect accesses (with possibility for reuse)

Indirect arguments, type double, dimensionality of 2

Direct arguments, type double, dimensionality of 4 and 1

Static code
Managing data for vectorization

Vertex data

Cell data

Edge data
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];
deltas[cell2edge[(i+1)*4+0]]+=edge1[0][1];
deltas[cell2edge[(i+2)*4+0]]+=edge1[0][2];
deltas[cell2edge[(i+3)*4+0]]+=edge1[0][3];
...
}

And even that doesn't work consistently...
```
Vector intrinsics

• Clearly, nobody wants to write such code:

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

\[ adt = \_\text{mm256}\text{add}\text{pd}(\_\text{mm256}\text{max}\text{pd}(\_\text{mm256}\text{sub}\text{pd}(\_\text{mm256}\text{mul}\text{pd}(u, dy), \_\text{mm256}\text{mul}\text{pd}(v, dx)), \_\text{mm256}\text{sub}\text{pd}(\_\text{mm256}\text{mul}\text{pd}(v, dx), \_\text{mm256}\text{mul}\text{pd}(u, dy))), \_\text{mm256}\text{mul}\text{pd}(c, \_\text{mm256}\text{sqrt}\text{pd}(\_\text{mm256}\text{add}\text{pd}(\_\text{mm256}\text{mul}\text{pd}(dx, dx), \_\text{mm256}\text{mul}\text{pd}(dy, dy)))))); \]

• 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+i, &cell2vertex[i*4], 2, 4),
                     doublev(coords+i+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

Dual socket Sandy Bridge Xeon E5-2640 188 GFLOPS 65 GB/s
CPU vectorized performance

Double(Single) precision breakdowns

<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>

Comparison: non-vectorized CPU & GPU

<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>
MPI+OpenMP gives better performance than the pure MPI approach. Unlike in the case of the CPU, the hybrid setup can exploit new features in the Phi, such as gather instructions, vector reduction, etc., and integrate it seamlessly into OpenCL applications.

Figure 7: Performance on the Xeon Phi when varying the number of OpenMP threads per process to give a total of 240 (60 cores, 4 threads per core), as the number of MPI processes varies, so does the total number of OpenMP threads. This has non-trivial effects on the cost of communications, shared cache space, NUMA effects on the device, etc. However, the Xeon Phi is quite sensitive to being fully utilized; while the problem size is very small, we found that the Xeon Phi performance is only 4.8 and 3.9 times, respectively, for small and large meshes, compared with 3.85 and 3.25 times more time in double precision to finish the execution for the vectorized MPI+OpenMP version. As we increased the problem size, further, the non-vectorized version speeds up by 2.5 in single and 1.95 in double precision, while the vectorized version speeds up by 2.97 times more in single and 2.37 times more in double precision.

Figure 8: Performance of the Xeon Phi on the 2.8M cell mesh compared to the 720k cell mesh. Similar to the CPU, vectorization gives a significant performance boost, but the difference is even bigger in double precision. Unlike in the CPU case, the hybrid MPI+OpenMP setup - how many MPI processes and OpenMP threads, setting the number of OpenMP threads per process to give a total of 240 (60 cores, 4 threads per core), as the number of MPI processes vary, so does the total number of OpenMP threads is 240 (60 cores, 4 threads per core), as the number of MPI processes vary, so does the total number of OpenMP threads. Tests included pure MPI execution and different combinations of MPI processes and OpenMP threads, setting the number of OpenMP threads per process to give a total of 240 (60 cores, 4 threads per core), as the number of MPI processes vary, so does the total number of OpenMP threads.

Figure 6 shows performance figures on the Xeon Phi, solving the 720k cell mesh. There are significant differences in performance between the pure MPI and the hybrid MPI+OpenMP configurations. The hybrid setup shows a performance boost compared to the pure MPI configuration, but the difference is even bigger in double precision. Unlike in the case of the CPU, the hybrid setup can exploit new features in the Phi, such as gather instructions, vector reduction, etc., and integrate it seamlessly into OpenCL applications.

An important factor affecting the performance is the hybrid MPI+OpenMP setup - how many MPI processes and OpenMP threads, setting the number of OpenMP threads per process to give a total of 240 (60 cores, 4 threads per core), as the number of MPI processes vary, so does the total number of OpenMP threads. Tests included pure MPI execution and different combinations of MPI processes and OpenMP threads, setting the number of OpenMP threads per process to give a total of 240 (60 cores, 4 threads per core), as the number of MPI processes vary, so does the total number of OpenMP threads.

Table 8: Timing and bandwidth breakdowns for the Xeon Phi benchmarks in double (single) precision using the vectorized MPI+OpenMP backend on the Xeon Phi. The table shows the effect of varying these parameters on the per-loop breakdowns for the Xeon Phi. The time and bandwidth (BW) breakdowns are shown for the vectorized MPI+OpenMP version and the non-vectorized version. The non-vectorized version performs unreasonably slow, as expected, and the vectorized version shows a significant performance boost, especially in double precision.

Figure 7: Performance on the Xeon Phi when varying the number of OpenMP threads per process to give a total of 240 (60 cores, 4 threads per core), as the number of MPI processes vary, so does the total number of OpenMP threads. Tests included pure MPI execution and different combinations of MPI processes and OpenMP threads, setting the number of OpenMP threads per process to give a total of 240 (60 cores, 4 threads per core), as the number of MPI processes vary, so does the total number of OpenMP threads.
Xeon Phi performance

Non-linear 2D inviscid airfoil code - 2.8M cells, 1000 iterations
**Xeon Phi performance**

**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>0.58(0.25)</td>
<td>80(90)</td>
</tr>
<tr>
<td>adt_calc</td>
<td>2.0(1.16)</td>
<td>57(50)</td>
</tr>
<tr>
<td>res_calc</td>
<td>7.52(5.47)</td>
<td>45(38)</td>
</tr>
<tr>
<td>update</td>
<td>2.55(1.93)</td>
<td>77(50)</td>
</tr>
</tbody>
</table>

For reference, on the CPU:

<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>
Performance summary

Non-linear 2D inviscid airfoil code - 2.8M cells, 1000 iterations
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>
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?