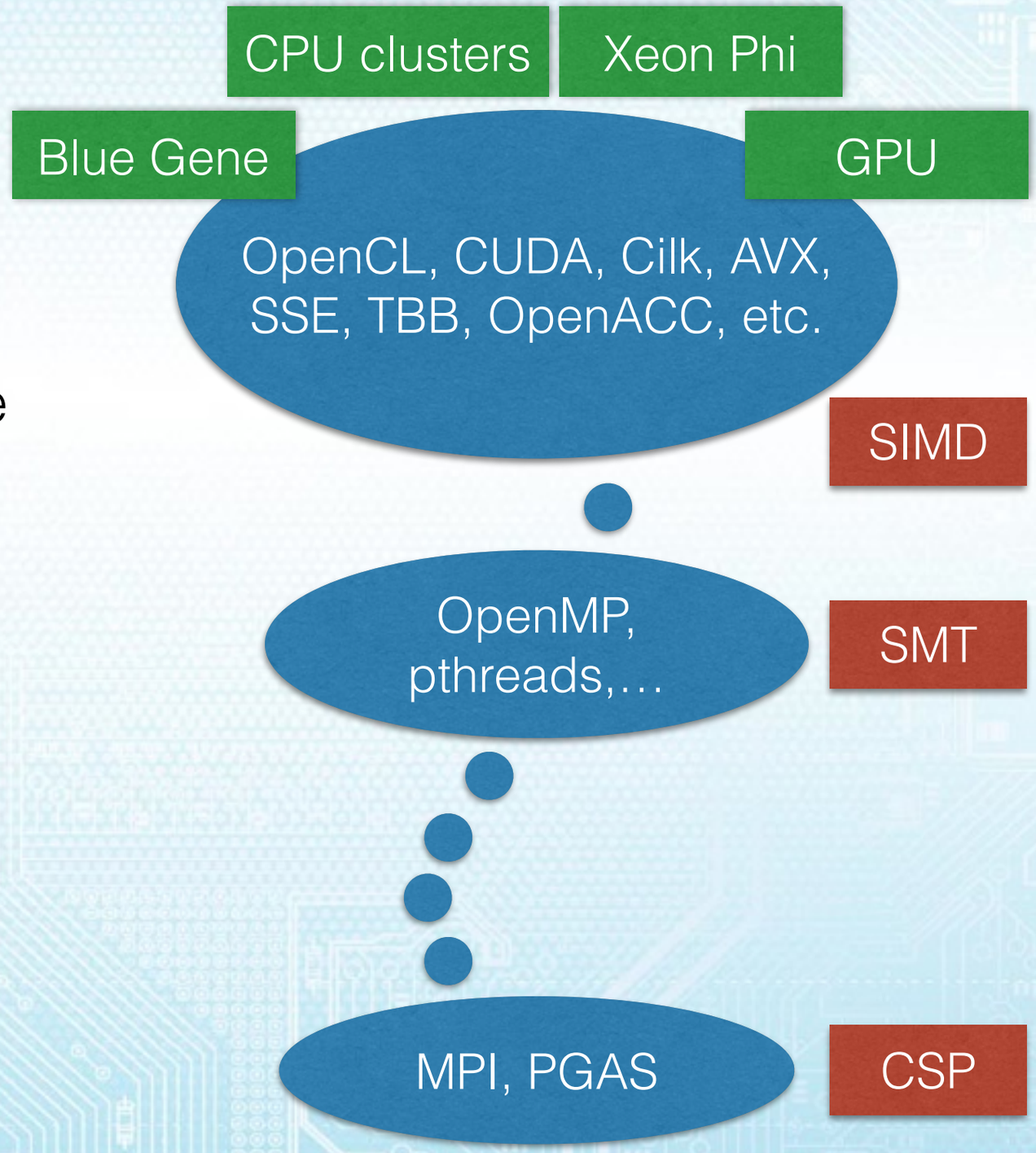
# Vectorizing Unstructured Mesh Computations for Many-core Architectures

*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



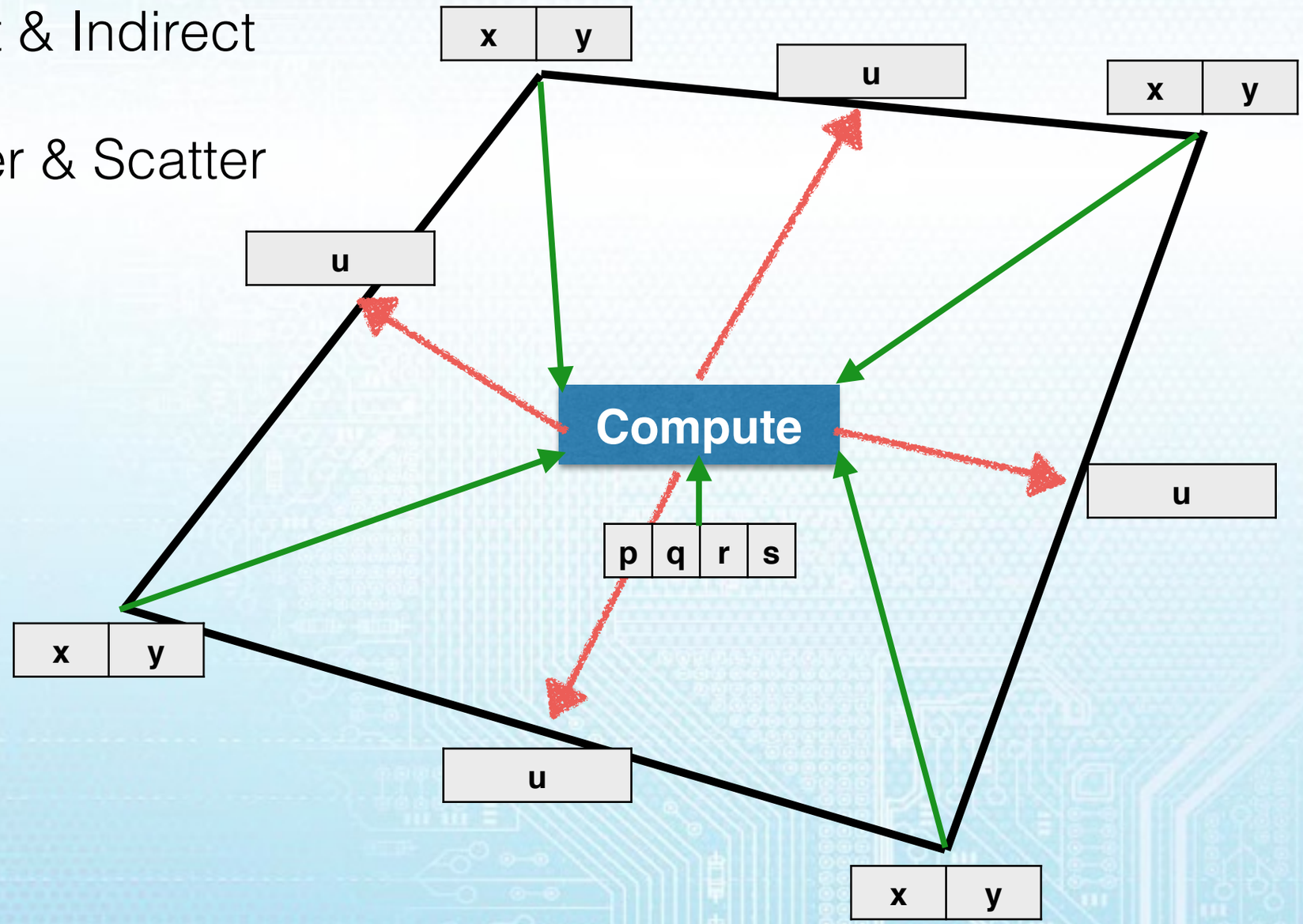
# Bridging the gap

- 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

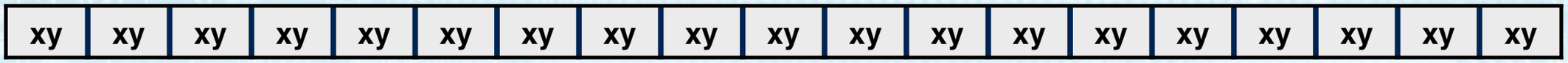
- Direct & Indirect
- Gather & Scatter



# 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

# Vertex data



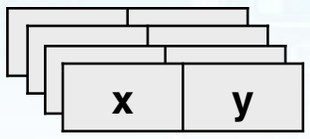

# Cell data



# Edge data

# Execution

Take the following (oversimplified) example



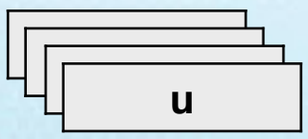




```
for (int i =0; i < set_size; i++) {
  double *n1 = &coords[2*cell2vertex[i*4+0]];
  double *n2 = &coords[2*cell2vertex[i*4+1]];
  double *n3 = &coords[2*cell2vertex[i*4+2]];
  double *n4 = &coords[2*cell2vertex[i*4+3]];
  double *ce = &celldata[4*i];
  double edge1[1],edge2[1],edge3[1],edge4[1];
  //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];
}
```

Indirect reads

Direct read

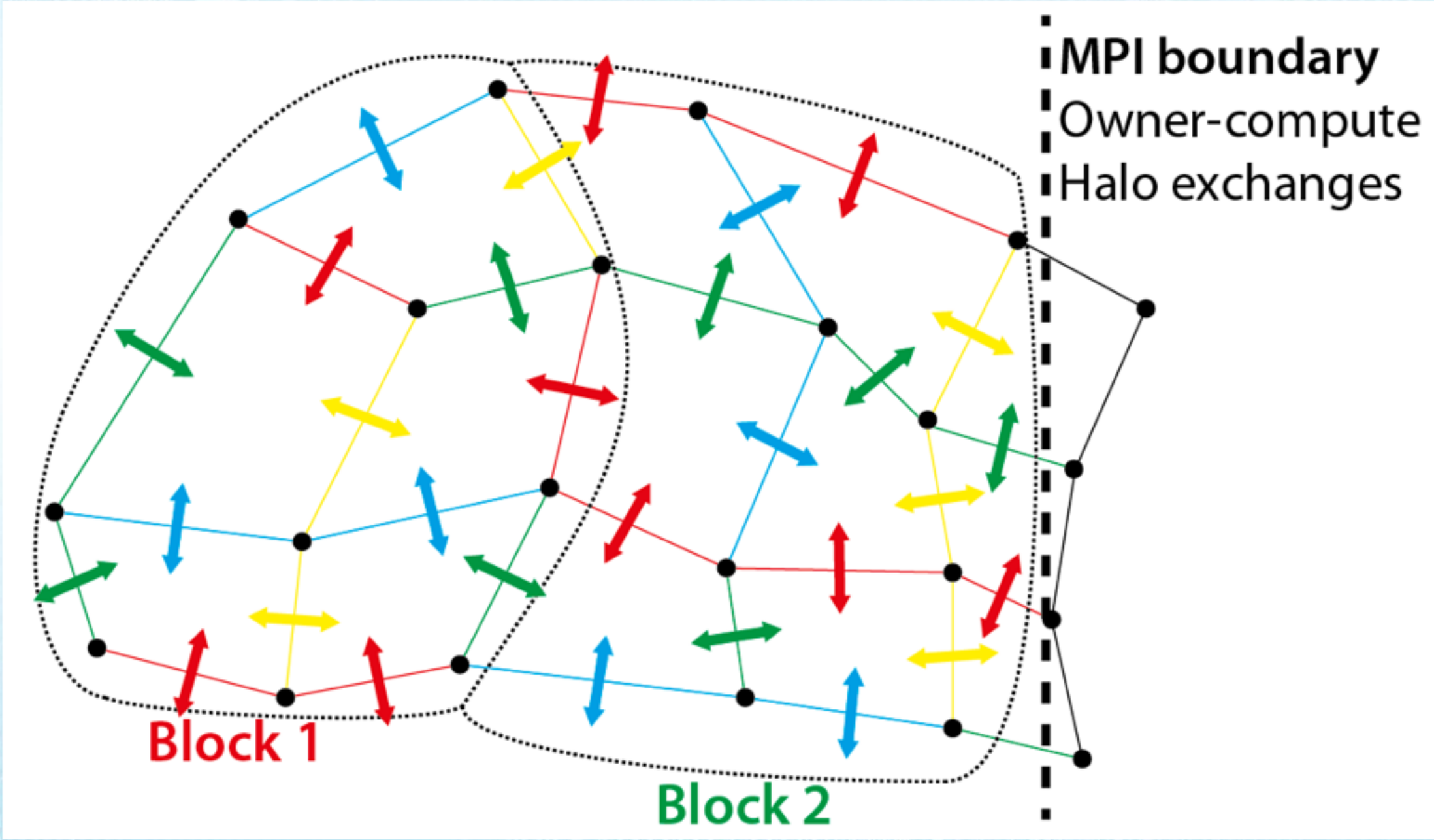



Indirect increments

# 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



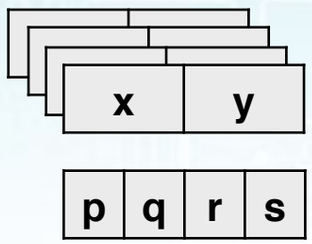
# CPU - coarse grained

- 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

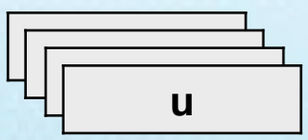
# Mapping to SIMT




```
__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];
    }
  }
}
```

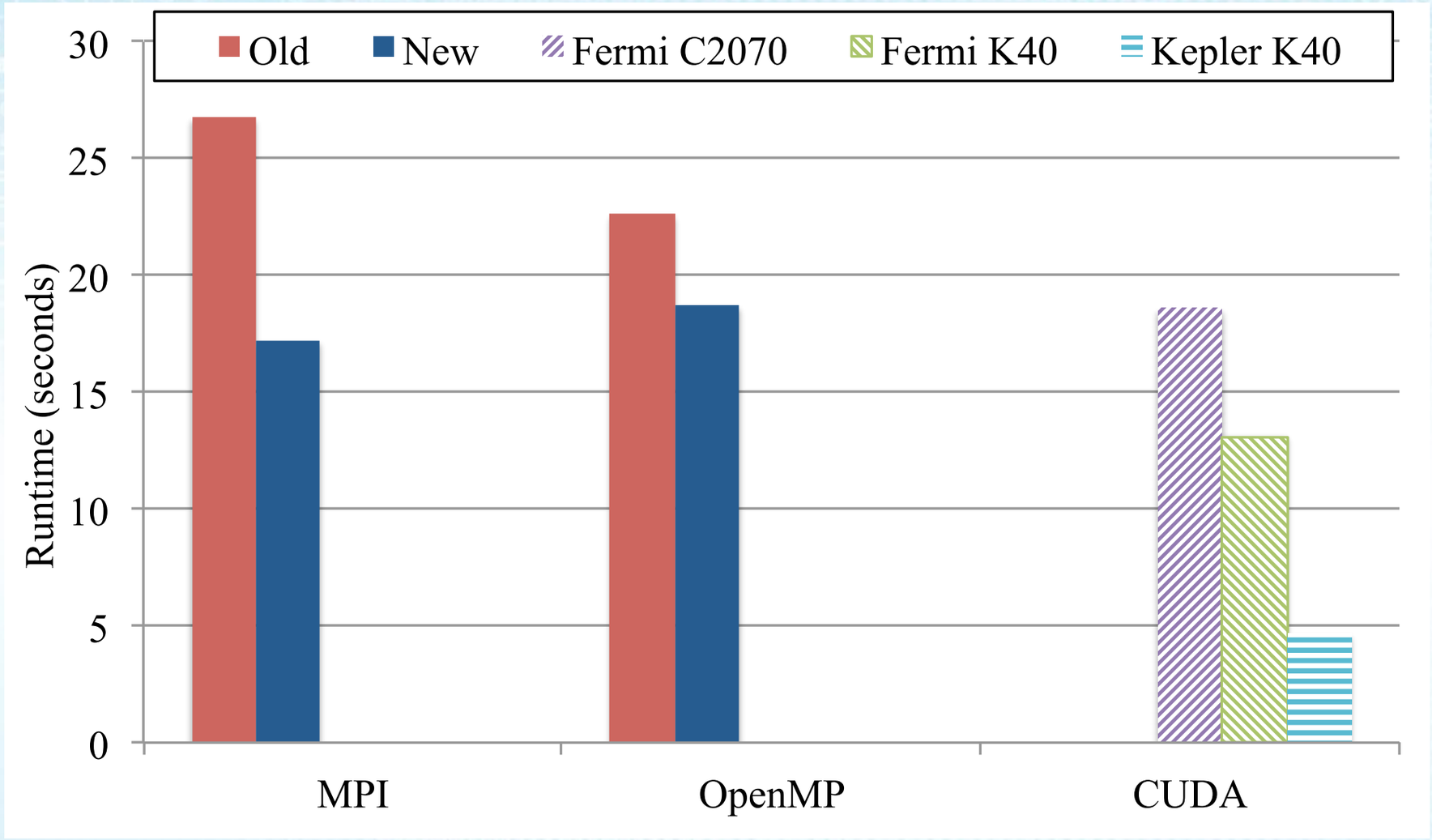
Indirect reads

Direct read




Indirect increments

# CPU and GPU baseline




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

| Dual socket Sandy Bridge Xeon E5-2640 | Tesla K40 |
|---|---|
| 188 GFLOPS | 2880 CUDA cores |
| 65 GB/s | 1420 GFLOPS |
| | 229 GB/s |

# CPU and GPU baseline

Bandwidth in GB/s and Compute in GFLOPS

| Kernel | pure MPI | | | CUDA | | |
|---|---|---|---|---|---|---|
| | Time | BW | Comp | Time | BW | Comp |
| save_soln | 0.99 | 46.55 | 2.9 | 0.20 | 230 | 14.4 |
| adt_calc | 6.3 | 18.24 | 14.6 | 0.71 | 161.2 | 129.2 |
| res_calc | 6.58 | 56.72 | 31.87 | 2.8 | 133.4 | 74.95 |
| bres_calc | 0.03 | 27.15 | 13.62 | 0.03 | 26.3 | 13.2 |
| update | 3.23 | 60.62 | 7.57 | 0.85 | 228 | 28.52 |

Direct →

Gather →

Gather & Scatter →

Direct →

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));
```

Number of arguments

Indirection property

Direct/indirect argument

Dataset dimensionality

Argument type

Access type

```
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

Static code

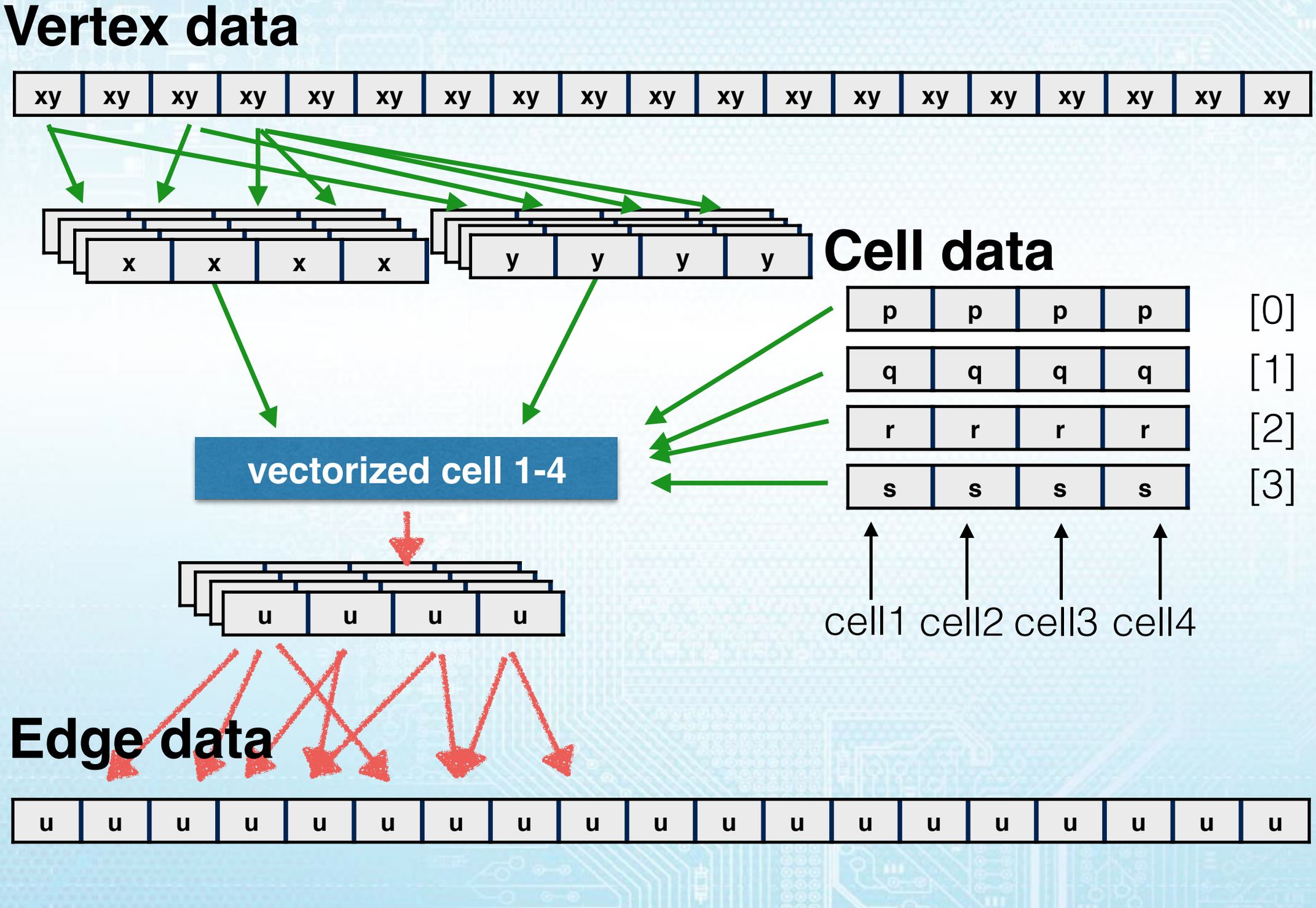
Setting up indices for indirect accesses (with possibility for reuse)

Indirect arguments, type double, dimensionality of 2

Direct arguments, type double, dimensionality of 4 and 1

Static code

# Managing data for vectorization



# Enabling vectorization

We have to spoon-feed the compiler

```
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];
  ...
}
```

} Gather

} Scatter

**And even that doesn’t work consistently…**

# Vector intrinsics

- Clearly, nobody wants to write such code:

```
 adt  = fabs(u*dy-v*dx) + c*sqrt(dx*dx+dy*dy);
 adt  =
_mm256_add_pd(_mm256_max_pd(_mm256_sub_pd(_mm256_mul_pd(u,dy),_mm256_mul_pd(v,dx)),_mm256_su
b_pd(_mm256_mul_pd(v,dx),_mm256_mul_pd(u,dy))),
_mm256_mul_pd(c,_mm256_sqrt_pd(_mm256_add_pd(_mm256_mul_pd(dx,dx),_mm256_mul_pd(dy,dy)))));
```

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

```
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:

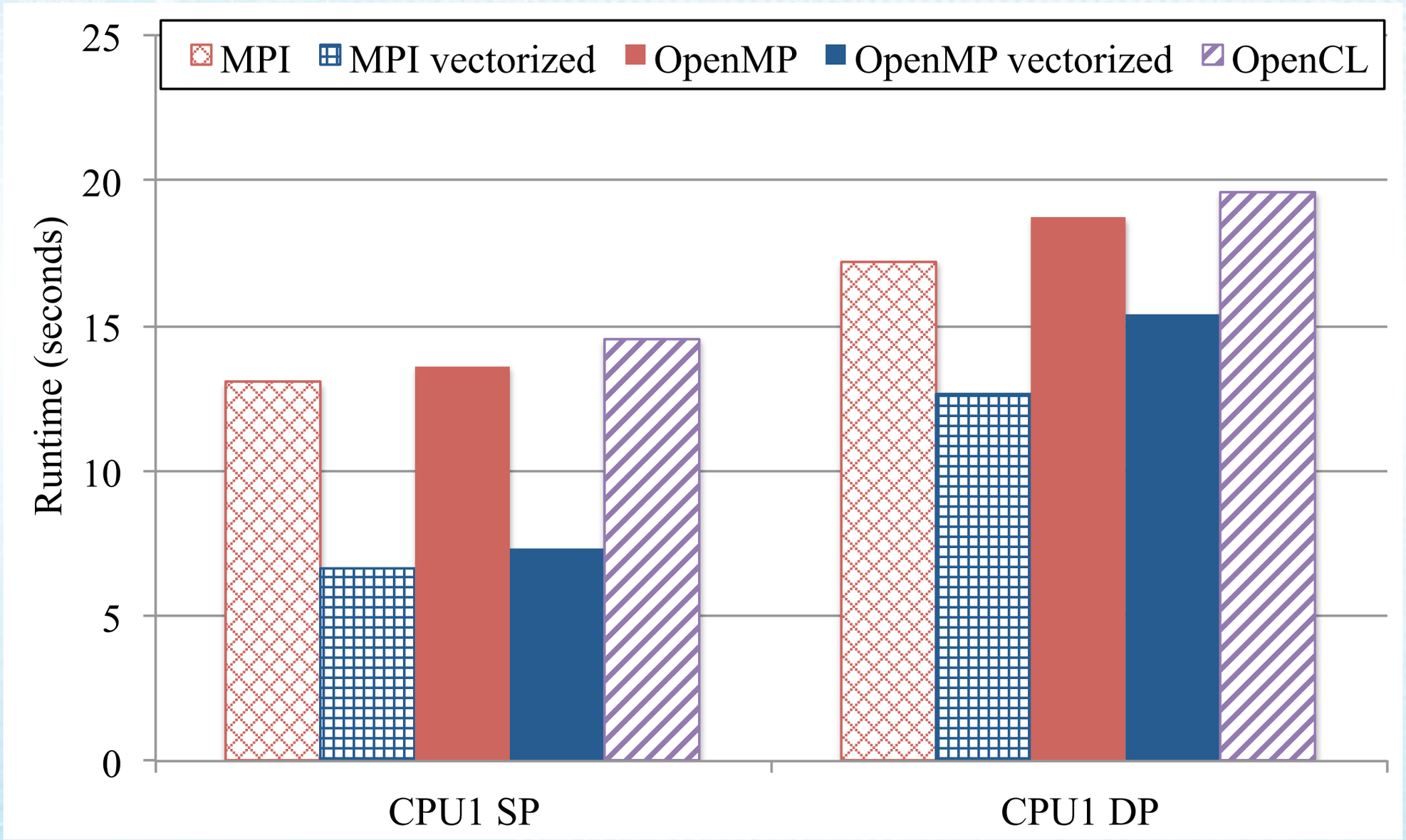
```
for (int i =0; i < set_size; i+=4) {
  doublev n1[2] ={doublev(coords+0, &cell2vertex[i*4], 2, 4),
                  doublev(coords+1, &cell2vertex[i*4], 2, 4)};
  ...

  //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




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

# CPU vectorized performance

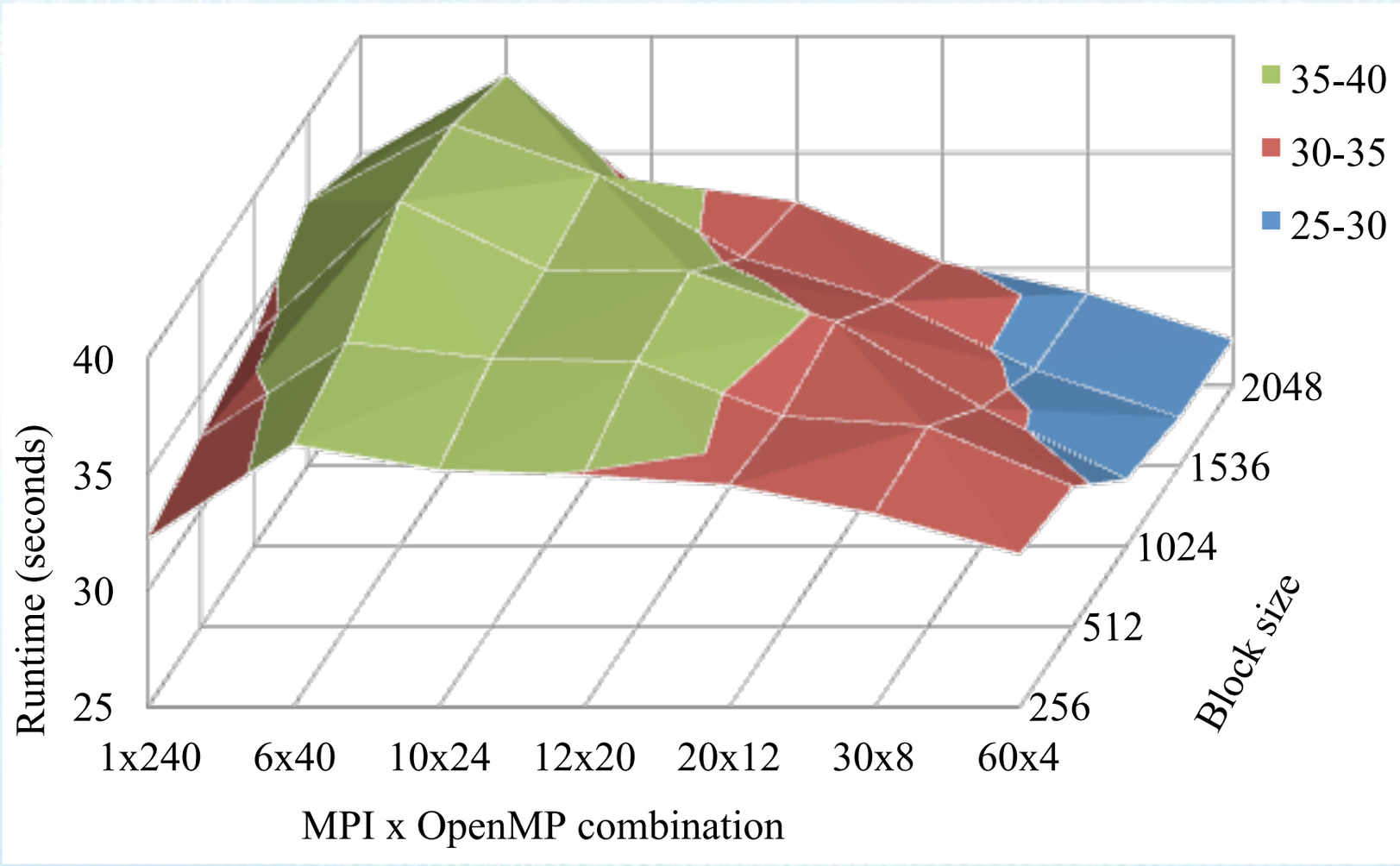
## Double(Single) precision breakdowns

| Kernel | 720k cells | | 2.8M cells | |
|---|---|---|---|---|
| | Time | BW | Time | BW |
| save_soln | 1.01(0.28) | 45(82) | 4.1(2.0) | 45(45) |
| adt_calc | 3.3(1.33) | 34(44) | 12.7(5.2) | 37(46) |
| res_calc | 5.06(3.5) | 73(59) | 19.5(13.5) | 76(62) |
| update | 3.33(1.5) | 59(65) | 14.6(7.0) | 54(56) |

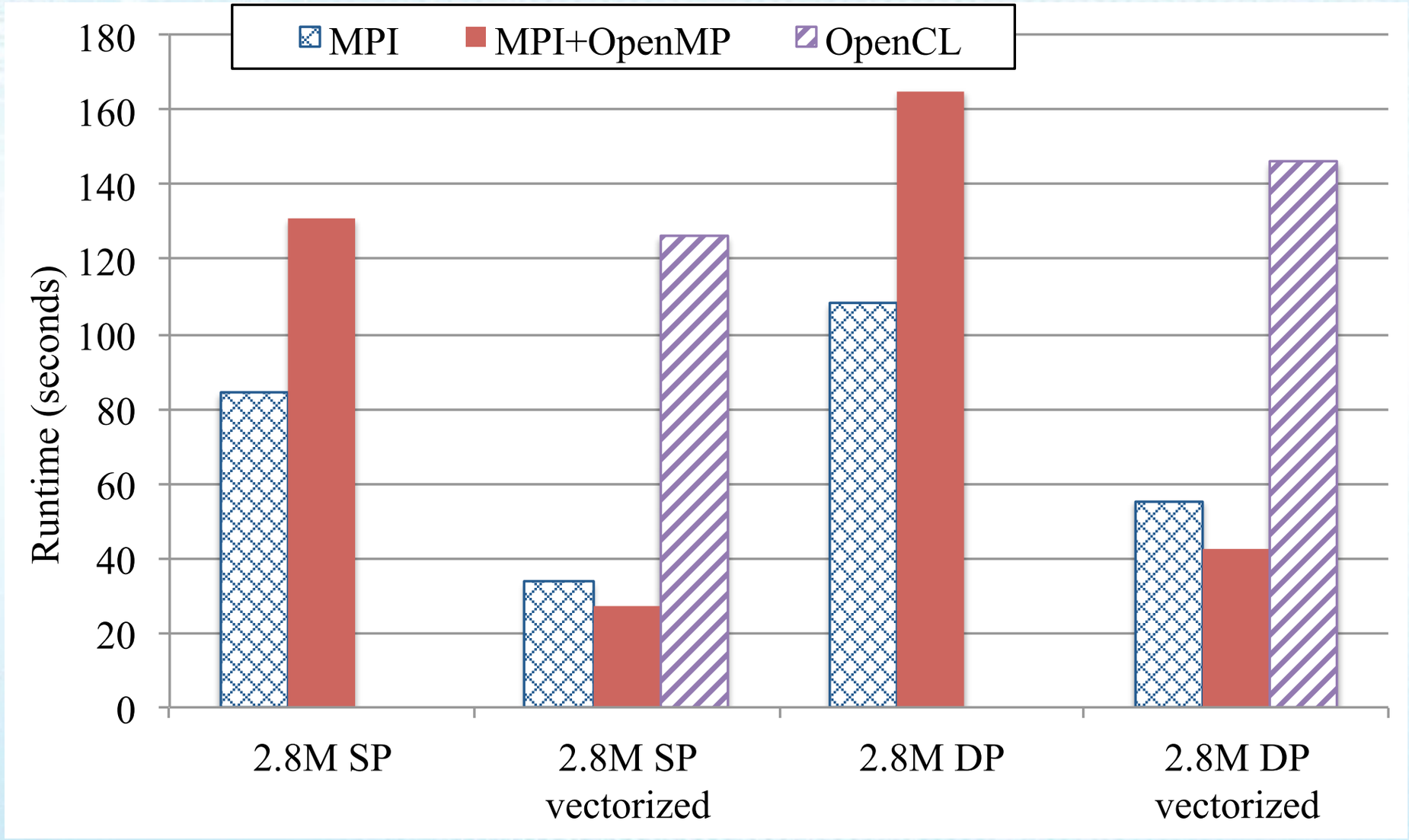
## Comparison: non-vectorized CPU & GPU

| Kernel | pure MPI | | | CUDA | | |
|---|---|---|---|---|---|---|
| | Time | BW | Comp | Time | BW | Comp |
| save_soln | 0.99 | 46.55 | 2.9 | 0.20 | 230 | 14.4 |
| adt_calc | 6.3 | 18.24 | 14.6 | 0.71 | 161.2 | 129.2 |
| res_calc | 6.58 | 56.72 | 31.87 | 2.8 | 133.4 | 74.95 |
| bres_calc | 0.03 | 27.15 | 13.62 | 0.03 | 26.3 | 13.2 |
| update | 3.23 | 60.62 | 7.57 | 0.85 | 228 | 28.52 |

# Xeon Phi - MPI+OpenMP



# Xeon Phi performance




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

# Xeon Phi performance

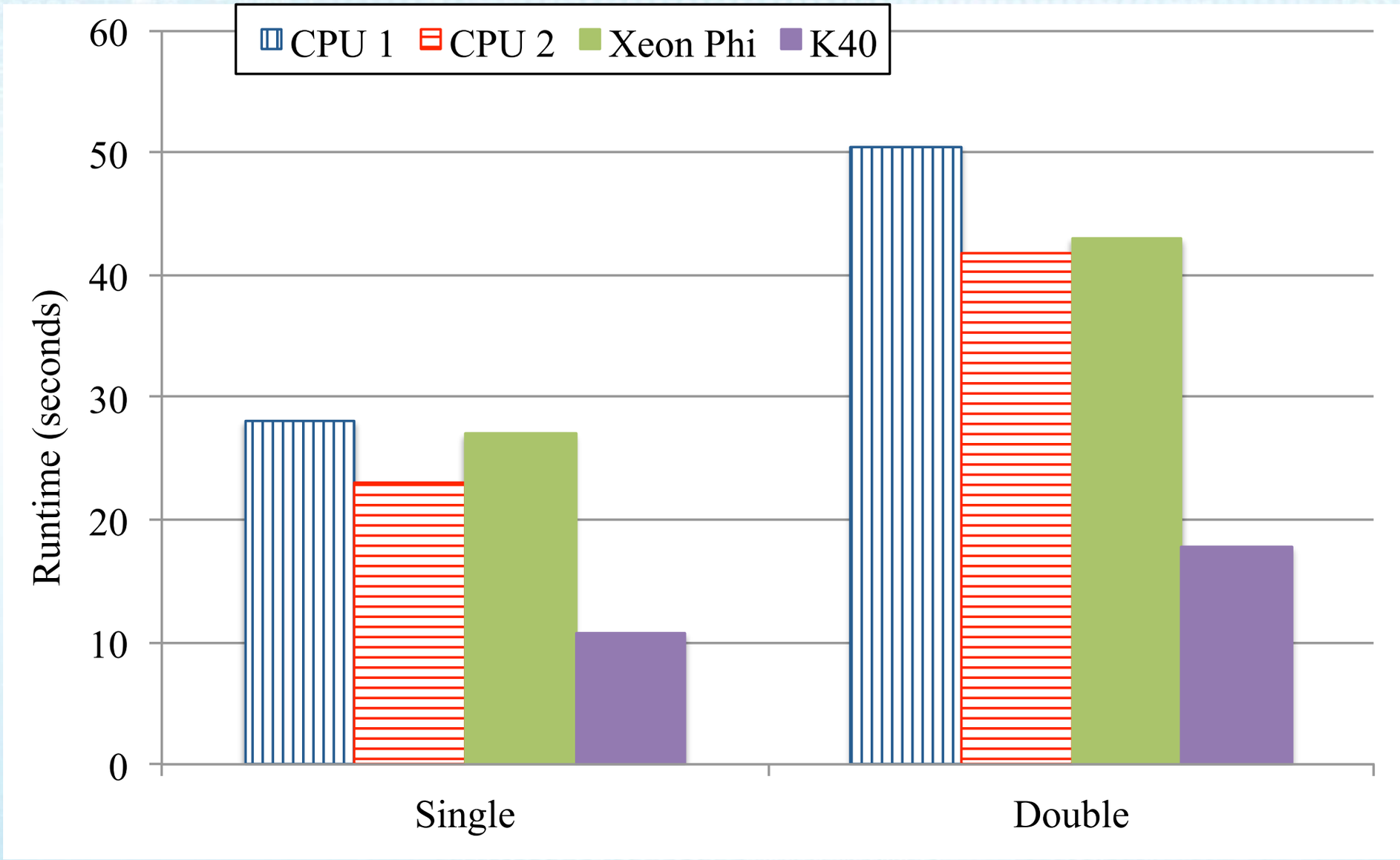
## Double(Single) precision breakdowns

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

## For reference, on the CPU:

| Kernel | 720k cells | | 2.8M cells | |
|---|---|---|---|---|
| | Time | BW | Time | BW |
| save_soln | 1.01(0.28) | 45(82) | 4.1(2.0) | 45(45) |
| adt_calc | 3.3(1.33) | 34(44) | 12.7(5.2) | 37(46) |
| res_calc | 5.06(3.5) | 73(59) | 19.5(13.5) | 76(62) |
| update | 3.33(1.5) | 59(65) | 14.6(7.0) | 54(56) |

# Performance summary




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

# Performance summary

Relative speedup over CPU 1 in double precision

| Kernel | CPU 1 | CPU 2 | Xeon Phi | K40 |
|---|---|---|---|---|
| save_soln | 1.0 | 1.04 | 1.88 | 5.11 |
| adt_calc | 1.0 | 1.43 | 1.87 | 4.67 |
| res_calc | 1.0 | 1.33 | 0.81 | 1.79 |
| update | 1.0 | 1.03 | 1.67 | 4.49 |

# 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?**