Initial Experiences in Porting a GPU Graph Analysis Workload to Chapel

Paul Sathre, Atharva Gondhalekar, & Wu-Chun Feng
June 2, 2023
Why GPU via Chapel?

*Productivity* for non-GPU-experts
How would you rather add vectors on a GPU?
How would you rather add vectors on a GPU?

OpenCL (via MetaCL)

```
__kernel void vecAddKernel(__global float *A, __global float *B, __global float *C, int nelem) {
  size_t tid = get_global_id(0);
  if (tid < nelem) {
  }
}

void vecAdd(float *A_h, float *B_h, float *C_h, int32_t nelem) {
  meta_set_acc(-1, metaModePreferOpenCL);
  cl_device_id dev;
  cl_platform_id plat;
  cl_context ctx;
  cl_command_queue q;
  meta_get_state_OpenCL(&plat, &dev, &ctx, &q);
  cl_mem A, B, C;
  A = clCreateBuffer(ctx, NULL, sizeof(float) * nelem, NULL, NULL);
  B = clCreateBuffer(ctx, NULL, sizeof(float) * nelem, NULL, NULL);
  C = clCreateBuffer(ctx, NULL, sizeof(float) * nelem, NULL, NULL);
  clEnqueueWriteBuffer(q, A, CL_FALSE, 0, sizeof(float) * nelem, A_h, 0, NULL, NULL);
  clEnqueueWriteBuffer(q, B, CL_TRUE, 0, sizeof(float) * nelem, B_h, 0, NULL, NULL);
  size_t local[3] = {256, 1, 1};
  size_t global[3] = {(nelem / local[0]) + (nelem % local[0] ? 1 : 0)) * local[0], 1, 1};
  metacl_vecAdd_vecAddKernel(q, &global, &local, NULL, false, NULL, &A, &B, &C, nelem);
  //Copy buffers
  clEnqueueReadBuffer(q, C, CL_TRUE, 0, sizeof(float) * nelem, C_h, 0, NULL, NULL);
  clFinish(q);
  //Release buffers
  clReleaseMemObject(A);
  clReleaseMemObject(B);
  clReleaseMemObject(C);
}
```
How would you rather add vectors on a GPU?

**OpenCL (via MetaCL)**

1. __kernel void vecAddKernel(__global float *A, __global float *B, __global float *C, int nelem) {
2.   size_t tid = get_global_id(0);
3.   if (tid < nelem) {
5.   }
6. }
7. void vecAdd(float *A_h, float *B_h, float *C_h, int32_t nelem) {
8.   meta_set_acc(-1, metaModePreferOpenCL);
9.   cl_device_id dev;
10.  cl_platform_id plat;
11.   cl_context ctx;
12.   cl_command_queue q;
13.   meta_get_state_OpenCL(&plat, &dev, &ctx, &q);
14.   cl_mem A, B, C;
15.   A = clCreateBuffer(ctx, NULL, sizeof(float) * nelem, NULL, NULL);
16.   B = clCreateBuffer(ctx, NULL, sizeof(float) * nelem, NULL, NULL);
17.   C = clCreateBuffer(ctx, NULL, sizeof(float) * nelem, NULL, NULL);
18.   clEnqueueWriteBuffer(q, A, CL_FALSE, 0, sizeof(float) * nelem, A_h, 0, NULL, NULL);
19.   clEnqueueWriteBuffer(q, B, CL_TRUE, 0, sizeof(float) * work, B_h, 0, NULL, NULL);
20.   size_t local[3] = {256, 1, 1};
21.   size_t global[3] = {(nelem / local[0]) + (nelem % local[0] ? 1 : 0) * local[0], 1, 1};
22.   metacl_vecAdd_vecAddKernel(q, &global, &local, NULL, false, NULL, &A, &B, &C, nelem);
23.   //Copy buffers
24.   clEnqueueReadBuffer(q, C, CL_TRUE, 0, sizeof(float) * work, C_h, 0, NULL, NULL);
25.   clEnqueueReadBuffer(q, B, CL_TRUE, 0, sizeof(float) * work, B_h, 0, NULL, NULL);
26.   clEnqueueReadBuffer(q, A, CL_TRUE, 0, sizeof(float) * work, A_h, 0, NULL, NULL);
27.   clFinish(q);
28.   //Release buffers
29.   clReleaseMemObject(A);
30.   clReleaseMemObject(B);
31.   clReleaseMemObject(C);
32. }

**SYCL**

1. void vecAdd(float *A_h, float *B_h, float *C_h, int32_t nelem) {
2.   sycl::queue myQueue;
3.   sycl::buffer<float> A(A_h, nelem, sycl::property::buffer::use_host_ptr());
4.   sycl::buffer<float> B(B_h, nelem, sycl::property::buffer::use_host_ptr());
5.   sycl::buffer<float> C(C_h, nelem, sycl::property::buffer::use_host_ptr());
6.   C.set_write_back(true);
7.   sycl::range<1> local{256};
8.   sycl::range<1> global{((nelem / local.get(0)) + (nelem % local.get(0) ? 1 : 0)) * local.get(0)};
9.   myQueue.submit([&](sycl::handler &cgh) { //GPU submit
10.      auto A_acc = A.get_access<sycl::access::mode::read>(cgh, sycl::range<1>{(size_t)nelem});
11.      auto B_acc = B.get_access<sycl::access::mode::read>(cgh, sycl::range<1>{(size_t)nelem});
12.      auto C_acc = C.get_access<sycl::access::mode::discard_write>(cgh, sycl::range<1>{(size_t)nelem});
13.      cgh.parallel_for(sycl::nd_range<1>{global, local}, [=](sycl::nd_item<1> tid_info) {
14.        size_t tid = tid_info.get_global_linear_id();
15.        if (tid < nelem) {
17.        }
18.      });
19.   });
20.   myQueue.wait();
21. }

How I learned and spent 10+ years

How I've been doing it recently (on FPGA)

Initial Experiences in Porting a GPU Graph Analysis Workload to Chapel

CHIUW’23 -- June 2, 2023
How would you rather add vectors on a GPU?

OpenCL (via MetaCL)

- How I learned and spent 10+ years

SYCL

- How I've been doing it recently (on FPGA)

CUDA

- How most GPU kernels are written

Initial Experiences in Porting a GPU Graph Analysis Workload to Chapel

CHIUW'23 -- June 2, 2023
How would you rather add vectors on a GPU?

**OpenCL (via MetaCL)**

- Least Programmable
- Portable
- NVIDIA
- AMD
- Intel
- FPGA
- Portable
- CHIUW’23

**SYCL**

- Portable
- NVIDIA
- AMD
- Intel
- FPGA
- Not Portable

**CUDA**

- Portable
- NVIDIA
- AMD
- Intel
- FPGA
- Portable

**Chapel**

- Portable
- NVIDIA
- AMD
- Intel
- FPGA
- Portable

---

Initial Experiences in Porting a GPU Graph Analysis Workload to Chapel

CHIUW’23 -- June 2, 2023
How would you rather add vectors on a GPU?

OpenCL (via MetaCL)

**SYCL**

1. void vecAdd(float *A_h, float *B_h, float *C_h, int32_t nelem) {
2. //variable declarations get here
3. sycl::queue q;
4. sycl::nd_range<1> nd_range(global_dim3(global_dim[0]), local_dim3(local_dim[0]));
5. sycl::range<1> grid_size(global_dim3(global_dim[0]), 1);
6. sycl::range<1> local_size(256, 1);
7. sycl::event E;
8. A_h = cgh.parallel_for(nd_range, E);
9. B_h = cgh.parallel_for(nd_range, E);
10. C_h = cgh.parallel_for(nd_range, E);
11. E.wait();
12. }

**CUDA**

1. __global__ void vecAdd(float *A_h, float *B_h, float *C_h, int32_t nelem) {
2. //variable declarations get here
3. cudaMemcpy(B, B_h, sizeof(float) * nelem, cudaMemcpyHostToDevice);
4. cudaMemcpy(C, C_h, sizeof(float) * nelem, cudaMemcpyHostToDevice);
5. cudaMemcpy(A, A_h, sizeof(float) * nelem, cudaMemcpyHostToDevice);
6. for (int i = 0; i < nelem; i++) {
8. }
9. cudaMemcpy(C_h, C, sizeof(float) * nelem, cudaMemcpyDeviceToHost);
10. cudaMemcpy(B_h, B, sizeof(float) * nelem, cudaMemcpyDeviceToHost);
11. }

**Kernel & Launch**

**OpenCL**

- Use OpenCL (via MetaCL)
- Portable

**SYCL**

- Portable

**CUDA**

- Not Portable

Chapel

- Portable

Initial Experiences in Porting a GPU Graph Analysis Workload to Chapel

CHIUW'23 -- June 2, 2023
How would you rather add vectors on a GPU?

OpenCL (via MetaCL)  

SYCL

CUDA

Chapel

How I learned and spent 10+ years

How I've been doing it recently (on FPGA)

How most GPU kernels are written

How we get to talk about today =)

---

Initial Experiences in Porting a GPU Graph Analysis Workload to Chapel

CHIUW'23 -- June 2, 2023
How would you rather add vectors on a GPU?

**OpenCL (via MetaCL)**

```
void vecAdd(float *A, float *B, float *C, int nelem) {
    for (int idx = 0; idx < nelem; ++idx) {
    }
}
```

**SYCL**

```
void __global__ void vecAddKernel(float *A, float *B, float *C, int nelem) {
    int tid = get_global_id(0);
}
```

**CUDA**

```
__global__ void __device__ vecAdd(float *A, float *B, float *C, int nelem) {
    int tid = blockIdx.x * blockDim.x + threadIdx.x;
}
```

**Chapel**

```
use GPU;

proc vecAdd(A_h, B_h, C_h) {
    A_h = A_h + B_h;
}
```

How I learned and spent 10+ years

How I’ve been doing it recently (on FPGA)

How most GPU kernels are written

How we get to talk about today =)
Why Graph Analysis on GPU via Chapel?

Understand Chapel’s programmability and GPU performance on irregular applications relative to proven approaches
Graph Workload: Edge-connected Jaccard similarity

- Intersection over union for all edge-connected pairs’ 1-hop neighborhoods
  - Essentially a batch of |E| set-intersections
  - Lots of indirect access

- Why do we care?
  - *Strength* of similarity between known pairs
  - *Proxy* for more complex intersection algo’s
    - Recommendations, Community detection, ...
  - But mostly: *How amenable is GPU Chapel to irregular algorithms?*

- What’s our approach? Port two existing (CUDA/SYCL) kernel pipelines
  - Edge-centric: homegrown, 1 pair per thread → good at near-hypersparse graphs
  - Vertex-centric: from *legacy* CuGraph, 3D, n=8 threads per pair → better on denser graphs

How is the *programmability*?
Write how you know *first*, then try to make idiomatic

- Still “Chapel how a C programmer would write it”
- Currently only used promoted-array kernels once (vertex-centric *Fill*), but more opportunities exist
  - Edge-centric reverse-edge preprocessing (*Scan*)
  - intersect-over-union array division (*Weights*)
- C-like generic compressed sparse row (CSR) structure could be revisited
- But still modest source-lines improvement
- C-style Chapel: 756 lines
  - Edge-centric: 150
  - Vertex-Centric: 230
  - Everything else: 376
  - CUDA: 1212
  - All Kernels: 395
  - Everything else: 817
  - Everything else: 817
Vertex-Centric: More complex

- Vertex-centric intersect didn’t initially GPU-ize

```chapel
var intersect : [0..<numEdges] atomic real(32);
forall Z in srcIters by gridDim.z*blockDim.z {
    forall Y in destIters by gridDim.y*blockDim.y {
        forall X in isectIters by gridDim.x*blockDim.x {
            ... // bin-search
            intersect[writeAddr].add(1.0);
        }
    }
}
```

VC-CPU-Naïve

(↑ pseudo-code, see Github for real)
Vertex-Centric: More complex

- Vertex-centric intersect didn’t initially GPU-ize
  - 3D grid/block → Chapel pre-1.31 currently only supports 1D forall
  - Had to linearize to 1D with a very large index space

```chapel
var intersect : [0..<numEdges] atomic real(32);
forall Z in srcIters by gridDim.z*blockDim.z {
  forall Y in destIters by gridDim.y*blockDim.y {
    forall X in isectIters by gridDim.x*blockDim.x {
      ... // bin-search
      intersect[writeAddr].add(1.0);
    }
  }
}
```

VC-CPU-Naïve

```chapel
var intersect : [0..<numEdges] atomic real(32);
forall id in srcIters*destIters*isectIters {
  var nd_id : 3*int = get_ND_ID(id);
  var zCount = nd_id(2);
  while (zCount < zMax) {
    var yCount = nd_id(1);
    while (yCount < yMax) {
      var xCount = nd_id(0);
      while (xCount < xMax) {
        ... // bin-search
        intersect[writeAddr].add(1.0);
        xCount += blockDim.x*blockDim.x;
      }
      yCount += blockDim.y*blockDim.y;
    }
    zCount += blockDim.z*blockDim.z;
  }
}
```

VC-CPU-Linearized

(↑ pseudo-code, see Github for real)
Vertex-Centric: More complex

- Vertex-centric intersect didn’t initially GPU-ize
  - 3D grid/block -> Chapel pre-1.31 currently only supports 1D forall
    - Had to linearize to 1D with a very large index space
  - for-by loops -> Chapel by clause doesn’t GPU-ize yet
    - GPU codes often increment by the thread count to keep co-executing threads aligned to memory
    - Had to replace with while-count

```
VC-CPU-Linearized
var intersect : [0..<numEdges] atomic real(32);
forall Z in srcIters by gridDim.z*blockDim.z {
  forall Y in destIters by gridDim.y*blockDim.y {
    forall X in isectIters by gridDim.x*blockDim.x {
      ... // bin-search
      intersect[writeAddr].add(1.0);
    }
  }
}
```

```
VC-CPU-Naïve
var intersect : [0..<numEdges] atomic real(32);
forall Z in srcIters by gridDim.z*blockDim.z {
  forall Y in destIters by gridDim.y*blockDim.y {
   forall X in isectIters by gridDim.x*blockDim.x {
      ... // bin-search
      intersect[writeAddr].add(1.0);
    }
  }
}
```

(↑ pseudo-code, see Github for real)
Vertex-Centric: More complex

- Vertex-centric intersect didn’t initially GPU-ize
  - 3D grid/block → Chapel pre-1.31 currently only supports 1D forall
    - Had to linearize to 1D with a very large index space
  - for-by loops → Chapel by clause doesn’t GPU-ize yet
    - GPU codes often increment by the thread count to keep co-executing threads aligned to memory
    - Had to replace with while-count
  - Accumulate via atomicAdd → Chapel atomics don’t GPU-ize yet
    - Had to call CUDA’s via extern C

VC-CPU-Naïve

```chapel
var intersect : [0..numEdges] atomic real(32);
forall Z in srcIters by gridDim.z*blockDim.z {
  forall Y in destIters by gridDim.y*blockDim.y {
    forall X in isectIters by gridDim.x*blockDim.x {
      ... // bin-search
      intersect[writeAddr].add(1.0);
    }
  }
}
```

VC-CPU-Linearized

```chapel
var intersect : [0..<numEdges] atomic real(32);
forall id in srcIters*destIters*isectIters {
  var nd_id : 3*int = get_ND_ID(id);
  var zCount = nd_id(2);
  while (zCount < zMax) {
    var yCount = nd_id(1);
    while (yCount < yMax) {
      var xCount = nd_id(0);
      while (xCount < xMax) {
        ... // bin-search
        intersect[writeAddr].add(1.0);
        xCount += gridDim.x*blockDim.x;
      }
      yCount += gridDim.y*blockDim.y;
    }
    zCount += gridDim.z*blockDim.z;
  }
}
```

VC-GPU

//replace atomic type .add() w/ extern call
ex_atomicAdd(c_ptrTo(intersect), writeAddr, 1.0)

↑ pseudo-code, see Github for real
Vertex-Centric: More complex

- Vertex-centric intersect didn’t initially GPU-ize
  - 3D grid/block → Chapel pre-1.31 currently only supports 1D forall
    - Had to linearize to 1D with a very large index space
  - for-by loops → Chapel by clause doesn’t GPU-ize yet
    - GPU codes often increment by the thread count to keep co-executing threads aligned to memory
    - Had to replace with while-count
  - Accumulate via atomicAdd → Chapel atomics don’t GPU-ize yet
    - Had to call CUDA’s via extern C

- But we were able to manage it
  - And Chapel mapped intermediate kernels to CPU → supporting incremental validation

VC-CPU-Naïve

```chapel
var intersect : [0..<numEdges] atomic real(32);
forall Z in srcIters by gridDim.z*blockDim.z {
  forall Y in destIters by gridDim.y*blockDim.y {
    forall X in isectIters by gridDim.x*blockDim.x {
      ... // bin-search
      intersect[writeAddr].add(1.0);
    }
  }
}
```

VC-CPU-Linearized

```chapel
var intersect : [0..<numEdges] atomic real(32);
forall id in srcIters*destIters*isectIters {
  var nd_id : 3*int = get_ND_ID(id);
  var zCount = nd_id(2);
  while (zCount < zMax) {
    var yCount = nd_id(1);
    while (yCount < yMax) {
      var xCount = nd_id(0);
      while (xCount < xMax) {
        ... // bin-search
        intersect[writeAddr].add(1.0);
        xCount += gridDim.x*blockDim.x;
      }
      yCount += gridDim.y*blockDim.y;
    }
    zCount += gridDim.z*blockDim.z;
  }
}
```

VC-GPU

```chapel
//replace atomic type .add() w/ extern call
ex_atomicAdd(c_ptrTo(intersect), writeAddr, 1.0)
```

(↑ pseudo-code, see Github for real)
How well did it perform?
Test Data

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Description</th>
<th>V</th>
<th>E</th>
<th>Avg</th>
<th>Range</th>
<th>Std. Dev.</th>
<th>Gini Index</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kmer_A2a</td>
<td>Protein k-mers</td>
<td>171M</td>
<td>361M</td>
<td>2.11</td>
<td>39</td>
<td>0.56</td>
<td>0.055</td>
</tr>
<tr>
<td>Europe_osm</td>
<td>European roads</td>
<td>50.9M</td>
<td>108M</td>
<td>2.12</td>
<td>12</td>
<td>0.48</td>
<td>0.085</td>
</tr>
<tr>
<td>Road_usa</td>
<td>US roads</td>
<td>23.9M</td>
<td>57.7M</td>
<td>2.41</td>
<td>8</td>
<td>0.93</td>
<td>0.211</td>
</tr>
<tr>
<td>Road-roadNet-CA</td>
<td>California roads</td>
<td>1.96M</td>
<td>5.52M</td>
<td>2.82</td>
<td>11</td>
<td>0.99</td>
<td>0.185</td>
</tr>
<tr>
<td>Road-roadNet-PA</td>
<td>Pennsylvania roads</td>
<td>1.09M</td>
<td>3.08M</td>
<td>2.83</td>
<td>8</td>
<td>1.02</td>
<td>0.188</td>
</tr>
<tr>
<td>Delaunay_n24</td>
<td>Random Triangulations</td>
<td>16.8M</td>
<td>101M</td>
<td>6.00</td>
<td>23</td>
<td>1.34</td>
<td>0.122</td>
</tr>
<tr>
<td>circuit5M</td>
<td>Large circuit</td>
<td>5.56M</td>
<td>54.0M</td>
<td>9.71</td>
<td>1.29M</td>
<td>577</td>
<td>0.577</td>
</tr>
<tr>
<td>Soc-LiveJournal1</td>
<td>Social network</td>
<td>4.85M</td>
<td>85.7M</td>
<td>17.7</td>
<td>20.3K</td>
<td>52.0</td>
<td>0.711</td>
</tr>
<tr>
<td>Wikipedia-20070206</td>
<td>Web page links</td>
<td>3.57M</td>
<td>84.8M</td>
<td>23.8</td>
<td>188K</td>
<td>255</td>
<td>0.759</td>
</tr>
<tr>
<td>GL7d19</td>
<td>Voroni differentials</td>
<td>1.96M</td>
<td>74.6M</td>
<td>38.2</td>
<td>134</td>
<td>6.73</td>
<td>0.088</td>
</tr>
<tr>
<td>dieiFilterV2real</td>
<td>Dielectric resonator</td>
<td>1.16M</td>
<td>47.4M</td>
<td>40.9</td>
<td>104</td>
<td>16.1</td>
<td>0.201</td>
</tr>
<tr>
<td>Sc-lidoor</td>
<td>Large door</td>
<td>952K</td>
<td>41.5M</td>
<td>43.6</td>
<td>14.8</td>
<td>183</td>
<td>0.183</td>
</tr>
<tr>
<td>Stokes</td>
<td>VLSI process sim.</td>
<td>11.4M</td>
<td>516M</td>
<td>45.1</td>
<td>1728</td>
<td>61.8</td>
<td>0.392</td>
</tr>
<tr>
<td>Sc-msdoor</td>
<td>Medium Door</td>
<td>416K</td>
<td>45.1</td>
<td>45.1</td>
<td>76</td>
<td>13.7</td>
<td>0.166</td>
</tr>
<tr>
<td>Ca-coauthors dblp</td>
<td>Coauthorship</td>
<td>540K</td>
<td>56.4</td>
<td>71</td>
<td>6329</td>
<td>66.2</td>
<td>0.544</td>
</tr>
<tr>
<td>Sociorkut</td>
<td>Social network</td>
<td>3.00M</td>
<td>213M</td>
<td>98.9</td>
<td>11.5K</td>
<td>272</td>
<td>0.750</td>
</tr>
<tr>
<td>Hollywood-2009</td>
<td>Costarring Actors</td>
<td>1.14M</td>
<td>113M</td>
<td>18.9</td>
<td>1.7K</td>
<td>491</td>
<td>0.155</td>
</tr>
</tbody>
</table>

Graph data and images CC-BY-4.0 from the SparseSuite Matrix Collection (https://sparse.tamu.edu/).
Preprocessed CSR binary files: https://chrec.cs.vt.edu/SYCL-Jaccard/HPEC22-Data/index.html

Initial Experiences in Porting a GPU Graph Analysis Workload to Chapel
CHIUW’23 -- June 2, 2023
Fallback of running GPU forall on CPU is functionally-portable

- Great feature for validation, but don’t expect GPU-tuned impl’s to be performance-portable to CPU
Fallback of running GPU `forall` on CPU is *functionally-portable*

- Great feature for validation, but don’t expect GPU-tuned impl’s to be *performance-portable* to CPU
- But if we remove the fallback CPU columns...

Relative Performance (KernelTotal)

Higher is better!
GPU Performance is pretty good right out of the box

Higher is better!

CPU: AMD Threadripper 3960X
GPU: Nvidia RTX 3090
CUDA: 11.6 / driver 510.108.03
Chapel: pre-1.31 (d7664c9d81)

Initial Experiences in Porting a GPU Graph Analysis Workload to Chapel
CHIUW'23 -- June 2, 2023

Chapel Version | EC-GPU | VC-GPU
GPU Performance is pretty good right out of the box

- Room for tuning on the sparse end
GPU Performance is pretty good right out of the box

- Room for tuning on the sparse end, but approaching parity on the denser end
GPU Performance is pretty good right out of the box

- Room for tuning on the sparse end, but approaching parity on the denser end
- Chapel outperforms CUDA on a few inputs → Future work: Why? → tune CUDA
What’s next?

Future Work
• Understand performance gap and gains vs. CUDA
• Make more Chapel-idiomatic
  – Kernels: use array promotion
  – Generic CSR: abstract base class
• Implement other JS algorithms / optimizations
  – Consider distribution

Wishlist
• Chapel version of clang-format
• A pragma or attribute to name my kernels (for profiling & other tools)
  – Instead of picking out the right chpl_gpu_kernel## from assembly
• Native 2D/3D loops
• Warp/wavefront & sync primitives
In Conclusion / Q&A

• We need more *programmable* GPU languages → Chapel looks good to this GPU dev!

• Chapel was reasonable to port to with a “C on GPUs” background
  – Kernels: Embarrassingly-parallel is easy, thread-collaboration less-so but achievable *today*

• GPU Chapel’s programmability is less verbose than CUDA
  – “Chapel like a C programmer”: kernels take 96% as many lines, and the whole program only 62%
  – Should improve as we familiarize → revisit kernels for promotion and generic CSR representation

• GPU Chapel’s performance is slightly slower than CUDA, but within a shout
  – Between 0.51x and 1.98x performance, geo-mean 0.87x

• I am excited about Chapel and would recommend as an easier route to custom GPU kernels
Rudimentary Explicit RTTI: How we did it (C-style)

Kernels and I/O func’s “ladder-resolve” like this

//Resolve one runtime `var` to `param` per overload
proc fooOnCSR(in h : CSR_handle) {
    //resolve 1st param
    if (h.var1) then fooOnCSR(h, true) else fooOnCSR(h, false);
}
private proc fooOnCSR(in h : CSR_handle, param parm1 : bool) {
    ... // resolve 2nd param
    ... // resolve 3rd..Nth params
private proc fooOnCSR(in h : CSR_handle, param parm1 : bool, ...
    param parmN : bool) {
    doFooOnCSR(CSR(parm1, ..., parmN), h);
}
//Generic worker function
private proc doFooOnCSR(type csr_type : unmanaged CSR(?), in h : CSR_handle) {
    assert(csr_type == h.desc, ...);
    // Do Foo
}

proc makeCSR(in desc : CSR_descriptor) : CSR_handle {
    ... // resolve and allocate concrete instantiation of CSR generic class
}

class CSR {
    // Chapel bool `param` members
    // to instantiate concrete type
    // Chapel parameterized arrays to
    // contain vertex, edge, weight data
}

record CSR_header {
    // Disk binary format of bitfield
    // runtime type descriptor flags
}

record CSR_descriptor {
    // Chapel bool `var` members
    // to describe runtime type
}

proc foo(...) {
    var myDesc = readHeader(file) : CSR_descriptor;
    var myCSR = makeCSR(myDesc);
    readArrays(file, myCSR);
    ...
}

record CSR_handle {
    var desc : CSR_descriptor;
    var data : c_void_ptr;
}
Rudimentary RTTI: How we’d redo it (Chapel/OO-style)

Kernels and I/O func’s “ladder-resolve” like this

```chapel
//Resolve one runtime `var` to `param` per overload
proc fooOnCSR(in h : CSR_base) {
    //resolve 1st param
    if (h.var1) then fooOnCSR(h, true) else fooOnCSR(h, false);
    private proc fooOnCSR(in h : CSR_base, param parm1 : bool) {
        ... // resolve 2nd param }
    ... // resolve 3rd..Nth params
    private proc fooOnCSR(in h : CSR_base, param parm1 : bool, ...
        param parmN : bool) {
        doFooOnCSR(CSR_arrays(parm1, ..., parmN), h);
    }

    //Generic worker function
    private proc doFooOnCSR(type csr_type : unmanaged CSR_arrays(?),
        in h : CSR_base) {
        assert(csr_type.var1 == h.var1, ...);
        // Do Foo
    }
}
```

```chapel
//Disk binary format of bitfield // runtime type descriptor flags
record CSR_header {
// Chapel bool `var` members // to instantiate concrete type
}

class CSR_base {
// Disk binary format of bitfield // runtime type descriptor flags
}

class CSR_arrays : CSR_base {
// Chapel parameterized arrays to contain vertex, edge, weight data
}
```

```chapel
proc foo(...) {
    var myDesc = readHeader(file) : CSR_base;
    var myCSR = makeCSR(myDesc);
    readArrays(file, myCSR);
    ...
}
```

```chapel
proc makeCSR(in desc : CSR_base) : unmanaged CSR_base {
    ... // resolve and allocate concrete instantiation of CSR_arrays generic class
}
```