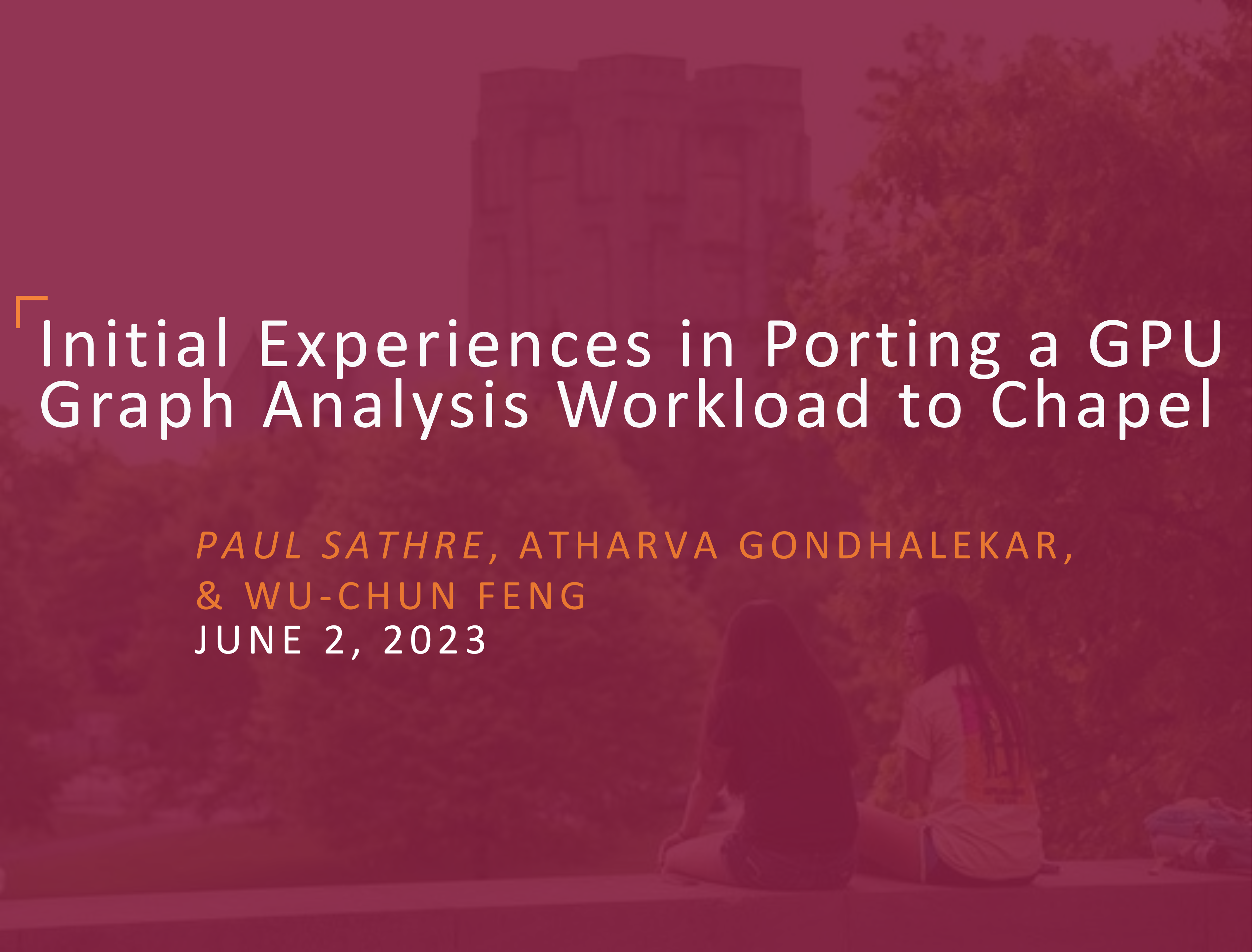


Initial Experiences in Porting a GPU Graph Analysis Workload to Chapel

PAUL SATHRE, ATHARVA GONDHALEKAR,
& WU-CHUN FENG

JUNE 2, 2023

VIRGINIA TECH

Why GPU via Chapel?

Productivity for non-GPU-experts

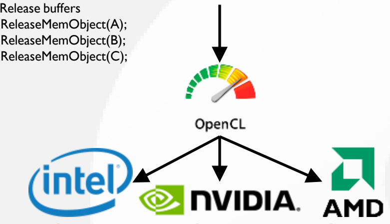
How would you rather add vectors on a GPU?

How would you rather add vectors on a GPU?

How I learned and
spent 10+ years

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) {
    C[tid] = A[tid] + B[tid];
  }
}
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) * work, 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) * work, C_h, 0, NULL, NULL);
  clFinish(q);
  //Release buffers
  clReleaseMemObject(A);
  clReleaseMemObject(B);
  clReleaseMemObject(C);
}
```



GPU
FPGA

GPU

GPU
FPGA

Portable

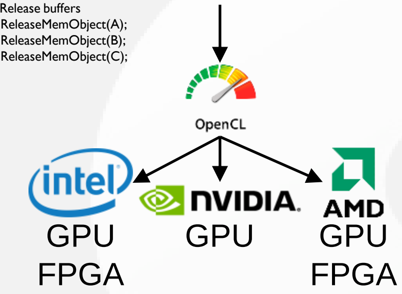
Least Programmable — Most Programmable

How would you rather add vectors on a GPU?

How I learned and
spent 10+ years

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) {
    C[tid] = A[tid] + B[tid];
  }
}
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) * work, 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) * work, C_h, 0, NULL, NULL);
  clFinish(q);
  //Release buffers
  clReleaseMemObject(A);
  clReleaseMemObject(B);
  clReleaseMemObject(C);
}
```

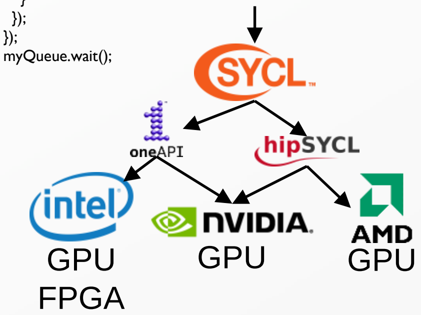


Portable

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

SYCL

```
void vecAdd(float *A_h, float *B_h, float *C_h, int32_t nelem) {
  sycl::queue myQueue;
  sycl::buffer<float> A(A_h, nelem, sycl::property::buffer::use_host_ptr());
  sycl::buffer<float> B(B_h, nelem, sycl::property::buffer::use_host_ptr());
  sycl::buffer<float> C(C_h, nelem, sycl::property::buffer::use_host_ptr());
  C.set_write_back(true);
  sycl::range<1> local{256};
  sycl::range<1> global{((nelem / local.get(0)) + (nelem % local.get(0) ? 1 : 0)) * local.get(0)};
  myQueue.submit([&](sycl::handler &cgh) { //GPU submit
    auto A_acc = A.get_access<sycl::access::mode::read>(cgh, sycl::range<1>{(size_t)nelem});
    auto B_acc = B.get_access<sycl::access::mode::read>(cgh, sycl::range<1>{(size_t)nelem});
    auto C_acc = C.get_access<sycl::access::mode::discard_write>(cgh, sycl::range<1>{(size_t)nelem});
    cgh.parallel_for(sycl::nd_range<1>{global, local}, [=](sycl::nd_item<1> tid_info) {
      size_t tid = tid_info.get_global_linear_id();
      if (tid < nelem) {
        C_acc[tid] = A_acc[tid] + B_acc[tid];
      }
    });
  });
  myQueue.wait();
}
```



Portable

Least Programmable

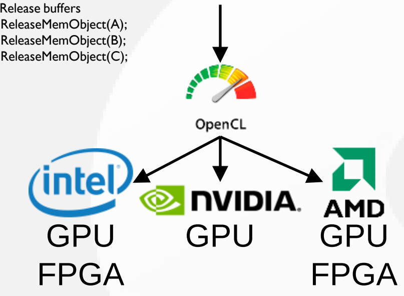
Most Programmable

How would you rather add vectors on a GPU?

How I learned and
spent 10+ years

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) {
    C[tid] = A[tid] + B[tid];
  }
}
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) * work, 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) * work, C_h, 0, NULL, NULL);
  clFinish(q);
  //Release buffers
  clReleaseMemObject(A);
  clReleaseMemObject(B);
  clReleaseMemObject(C);
}
```

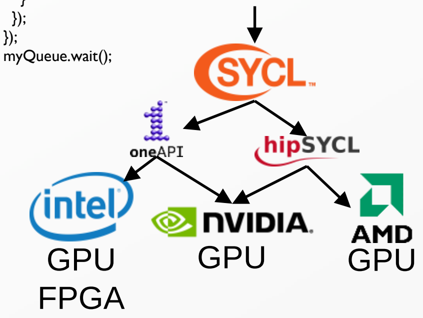


Portable

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

SYCL

```
void vecAdd(float *A_h, float *B_h, float *C_h, int32_t nelem) {
  sycl::queue myQueue;
  sycl::buffer<float> A(A_h, nelem, sycl::property::buffer::use_host_ptr());
  sycl::buffer<float> B(B_h, nelem, sycl::property::buffer::use_host_ptr());
  sycl::buffer<float> C(C_h, nelem, sycl::property::buffer::use_host_ptr());
  C.set_write_back(true);
  sycl::range<1> local{256};
  sycl::range<1> global{((nelem / local.get(0)) + (nelem % local.get(0) ? 1 : 0)) * local.get(0)};
  myQueue.submit([&](sycl::handler &cgh) { //GPU submit
    auto A_acc = A.get_access<sycl::access::mode::read>(cgh, sycl::range<1>{(size_t)nelem});
    auto B_acc = B.get_access<sycl::access::mode::read>(cgh, sycl::range<1>{(size_t)nelem});
    auto C_acc = C.get_access<sycl::access::mode::discard_write>(cgh, sycl::range<1>{(size_t)nelem});
    cgh.parallel_for(sycl::nd_range<1>{global, local}, [=](sycl::nd_item<1> tid_info) {
      size_t tid = tid_info.get_global_linear_id();
      if (tid < nelem) {
        C_acc[tid] = A_acc[tid] + B_acc[tid];
      }
    });
  });
  myQueue.wait();
}
```

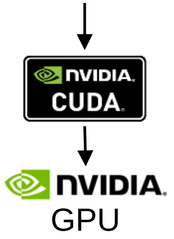


Portable

How **most** GPU
kernels are written

CUDA

```
__global__ void vecAddKernel(float *A, float *B, float *C, int32_t nelem) {
  size_t tid = blockDim.x * blockIdx.x + threadIdx.x;
  if (tid < nelem) {
    C[tid] = A[tid] + B[tid];
  }
}
void vecAdd(float *A_h, float *B_h, float *C_h, int32_t nelem) {
  float *A, *B, *C;
  int32_t work = hi-lo+1;
  cudaMalloc(&A, sizeof(float) * nelem);
  cudaMalloc(&B, sizeof(float) * nelem);
  cudaMalloc(&C, sizeof(float) * nelem);
  cudaMemcpy(A, A_h, sizeof(float) * nelem, cudaMemcpyHostToDevice);
  cudaMemcpy(B, B_h, sizeof(float) * nelem, cudaMemcpyHostToDevice);
  dim3 block = {256, 1, 1};
  dim3 grid = {(nelem / block.x) + (nelem % block.x ? 1 : 0), 1, 1};
  vecAddKernel<<<grid, block>>>(A, B, C, nelem);
  cudaMemcpy(C_h, C, sizeof(float) * nelem, cudaMemcpyDeviceToHost);
  cudaFree(dA);
  cudaFree(dB);
  cudaFree(dC);
}
```



Not
Portable

Least Programmable

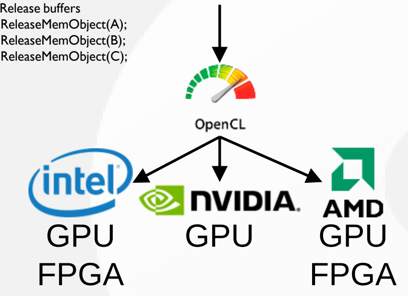
Most Programmable

How would you rather add vectors on a GPU?

How I learned and
spent 10+ years

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) {
    C[tid] = A[tid] + B[tid];
  }
}
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) * work, 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) * work, C_h, 0, NULL, NULL);
  clFinish(q);
  //Release buffers
  clReleaseMemObject(A);
  clReleaseMemObject(B);
  clReleaseMemObject(C);
}
```

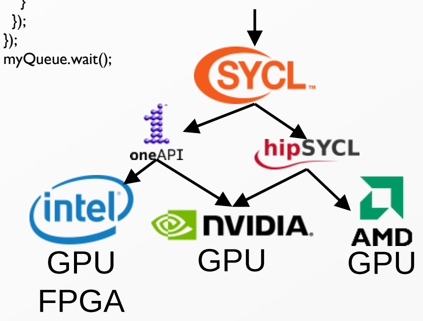


Portable

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

SYCL

```
void vecAdd(float *A_h, float *B_h, float *C_h, int32_t nelem) {
  sycl::queue myQueue;
  sycl::buffer<float> A(A_h, nelem, sycl::property::buffer::use_host_ptr());
  sycl::buffer<float> B(B_h, nelem, sycl::property::buffer::use_host_ptr());
  sycl::buffer<float> C(C_h, nelem, sycl::property::buffer::use_host_ptr());
  C.set_write_back(true);
  sycl::range<1> local{256};
  sycl::range<1> global{((nelem / local.get(0)) + (nelem % local.get(0) ? 1 : 0)) * local.get(0)};
  myQueue.submit([&](sycl::handler &cgh) { //GPU submit
    auto A_acc = A.get_access<sycl::access::mode::read>(cgh, sycl::range<1>{(size_t)nelem});
    auto B_acc = B.get_access<sycl::access::mode::read>(cgh, sycl::range<1>{(size_t)nelem});
    auto C_acc = C.get_access<sycl::access::mode::discard_write>(cgh, sycl::range<1>{(size_t)nelem});
    cgh.parallel_for(sycl::nd_range<1>{global, local}, [=](sycl::nd_item<1> tid_info) {
      size_t tid = tid_info.get_global_linear_id();
      if (tid < nelem) {
        C_acc[tid] = A_acc[tid] + B_acc[tid];
      }
    });
  });
  myQueue.wait();
}
```

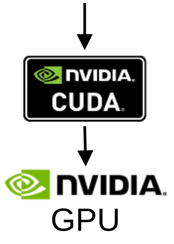


Portable

How **most** GPU
kernels are written

CUDA

```
__global__ void vecAddKernel(float *A, float *B, float *C, int32_t nelem) {
  size_t tid = blockDim.x * blockIdx.x + threadIdx.x;
  if (tid < nelem) {
    C[tid] = A[tid] + B[tid];
  }
}
void vecAdd(float *A_h, float *B_h, float *C_h, int32_t nelem) {
  float *A, *B, *C;
  int32_t work = hi-lo+1;
  cudaMalloc(&A, sizeof(float) * nelem);
  cudaMalloc(&B, sizeof(float) * nelem);
  cudaMalloc(&C, sizeof(float) * nelem);
  cudaMemcpy(A, A_h, sizeof(float) * nelem, cudaMemcpyHostToDevice);
  cudaMemcpy(B, B_h, sizeof(float) * nelem, cudaMemcpyHostToDevice);
  dim3 block = {256, 1, 1};
  dim3 grid = {(nelem / block.x) + (nelem % block.x ? 1 : 0), 1, 1};
  vecAddKernel<<<grid, block>>>(A, B, C, nelem);
  cudaMemcpy(C_h, C, sizeof(float) * nelem, cudaMemcpyDeviceToHost);
  cudaFree(dA);
  cudaFree(dB);
  cudaFree(dC);
}
```



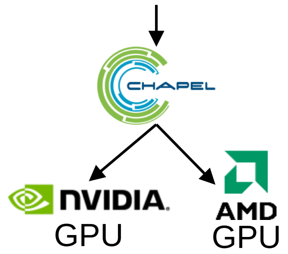
Not
Portable

OR

How we get to talk
about today =)

Chapel

```
use GPU;
proc vecAdd(A_h: [] real(32),
    B_h: [] real(32), C_h: [] real(32)) {
  on here.gpus[0] { //GPU locale
    var A: [A_h.domain] real(32);
    var B: [B_h.domain] real(32);
    var C: [C_h.domain] real(32);
    A = A_h; //copy-to
    B = B_h; //copy-to
    C = A + B; //GPU add
    C_h = C; //copy-from
  }
}
```



Portable

Least Programmable

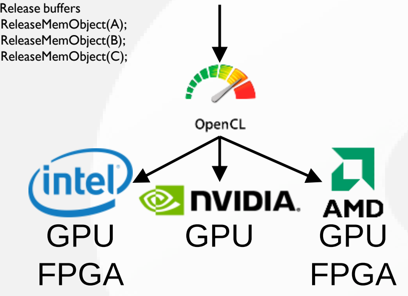
Most Programmable

How would you rather add vectors on a GPU?

How I learned and
spent 10+ years

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) {
    C[tid] = A[tid] + B[tid];
  }
}
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) * work, 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) * work, C_h, 0, NULL, NULL);
  clFinish(q);
  //Release buffers
  clReleaseMemObject(A);
  clReleaseMemObject(B);
  clReleaseMemObject(C);
}
```

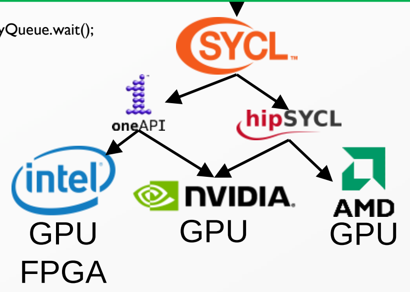


Portable

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

SYCL

```
void vecAdd(float *A_h, float *B_h, float *C_h, int32_t nelem) {
  sycl::queue myQueue;
  sycl::buffer<float> A(A_h, nelem, sycl::property::buffer::use_host_ptr());
  sycl::buffer<float> B(B_h, nelem, sycl::property::buffer::use_host_ptr());
  sycl::buffer<float> C(C_h, nelem, sycl::property::buffer::use_host_ptr());
  C.set_write_back(true);
  sycl::range<1> local{256};
  sycl::range<1> global{((nelem / local.get(0)) + (nelem % local.get(0) ? 1 :
0)) * local.get(0)};
  myQueue.submit([&](sycl::handler &cgh) { //GPU submit
    auto A_acc = A.get_access<sycl::access::mode::read>(cgh,
sycl::range<1>{(size_t)nelem});
    auto B_acc = B.get_access<sycl::access::mode::read>(cgh,
sycl::range<1>{(size_t)nelem});
    auto C_acc = C.get_access<sycl::access::mode::discard_write>(cgh,
sycl::range<1>{(size_t)nelem});
    cgh.parallel_for(sycl::nd_range<1>{global, local}, [=](sycl::nd_item<1>
tid_info) {
      size_t tid = tid_info.get_global_linear_id();
      if (tid < nelem) {
        C_acc[tid] = A_acc[tid] + B_acc[tid];
      }
    });
  });
  myQueue.wait();
}
```

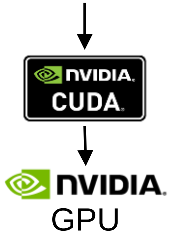


Portable

How **most** GPU
kernels are written

CUDA

```
__global__ void vecAddKernel(float *A, float *B, float *C, int32_t nelem) {
  size_t tid = blockDim.x * blockIdx.x + threadIdx.x;
  if (tid < nelem) {
    C[tid] = A[tid] + B[tid];
  }
}
void vecAdd(float *A_h, float *B_h, float *C_h, int32_t nelem) {
  float *A, *B, *C;
  int32_t work = hi-lo+1;
  cudaMalloc(&A, sizeof(float) * nelem);
  cudaMalloc(&B, sizeof(float) * nelem);
  cudaMalloc(&C, sizeof(float) * nelem);
  cudaMemcpy(A, A_h, sizeof(float) * nelem, cudaMemcpyHostToDevice);
  cudaMemcpy(B, B_h, sizeof(float) * nelem, cudaMemcpyHostToDevice);
  dim3 block = {256, 1, 1};
  dim3 grid = {(nelem / block.x) + (nelem % block.x ? 1 : 0), 1, 1};
  vecAddKernel<<<grid, block>>>(A, B, C, nelem);
  cudaMemcpy(C_h, C, sizeof(float) * nelem, cudaMemcpyDeviceToHost);
  cudaFree(dA);
  cudaFree(dB);
  cudaFree(dC);
}
```



Not
Portable

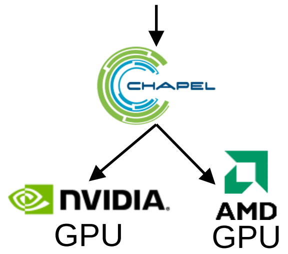
OR

Kernel &
Launch

How we get to talk
about today =)

Chapel

```
use GPU;
proc vecAdd(A_h: [] real(32),
  B_h: [] real(32), C_h: [] real(32)) {
  on here.gpus[0] { //GPU locale
    var A: [A_h.domain] real(32);
    var B: [B_h.domain] real(32);
    var C: [C_h.domain] real(32);
    A = A_h; //copy-to
    B = B_h; //copy-to
    C = A + B; //GPU add    (promoted)
    C_h = C; //copy-from
  }
}
```



Portable

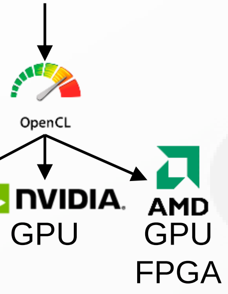
Least Programmable   Most Programmable

How would you rather add vectors on a GPU?

How I learned and
spent 10+ years

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) {
    C[tid] = A[tid] + B[tid];
  }
}
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) * work, 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) * work, C_h, 0, NULL, NULL);
  clFinish(q);
  //Release buffers
  clReleaseMemObject(A);
  clReleaseMemObject(B);
  clReleaseMemObject(C);
}
```



GPU FPGA | GPU | GPU FPGA

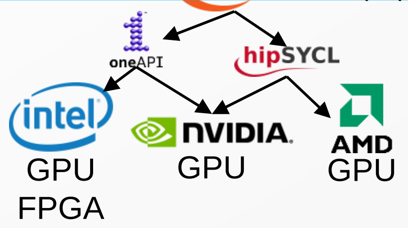
Portable

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

SYCL

```
void vecAdd(float *A_h, float *B_h, float *C_h, int32_t nelem) {
  sycl::queue myQueue;
  sycl::buffer<float> A(A_h, nelem, sycl::property::buffer::use_host_ptr());
  sycl::buffer<float> B(B_h, nelem, sycl::property::buffer::use_host_ptr());
  sycl::buffer<float> C(C_h, nelem, sycl::property::buffer::use_host_ptr());
  C.set_write_back(true);
  sycl::range<1> local{256};
  sycl::range<1> global{((nelem / local.get(0)) + (nelem % local.get(0) ? 1 :
0)) * local.get(0)};
  myQueue.submit([&](sycl::handler &cgh) { //GPU submit
    auto A_acc = A.get_access<sycl::access::mode::read>(cgh,
sycl::range<1>{(size_t)nelem});
    auto B_acc = B.get_access<sycl::access::mode::read>(cgh,
sycl::range<1>{(size_t)nelem});
    auto C_acc = C.get_access<sycl::access::mode::discard_write>(cgh,
sycl::range<1>{(size_t)nelem});
    cgh.parallel_for(sycl::nd_range<1>{global, local}, [=](sycl::nd_item<1>
tid_info) {
      size_t tid = tid_info.get_global_linear_id();
      if (tid < nelem) {
        C_acc[tid] = A_acc[tid] + B_acc[tid];
      }
    });
  });
  myQueue.wait();
}
```

(implicit)



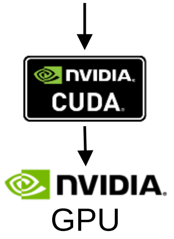
GPU FPGA | GPU | GPU

Portable

How **most** GPU
kernels are written

CUDA

```
__global__ void vecAddKernel(float *A, float *B, float *C, int32_t nelem) {
  size_t tid = blockDim.x * blockIdx.x + threadIdx.x;
  if (tid < nelem) {
    C[tid] = A[tid] + B[tid];
  }
}
void vecAdd(float *A_h, float *B_h, float *C_h, int32_t nelem) {
  float *A, *B, *C;
  int32_t work = hi-lo+1;
  cudaMalloc(&A, sizeof(float) * nelem);
  cudaMalloc(&B, sizeof(float) * nelem);
  cudaMalloc(&C, sizeof(float) * nelem);
  cudaMemcpy(A, A_h, sizeof(float) * nelem, cudaMemcpyHostToDevice);
  cudaMemcpy(B, B_h, sizeof(float) * nelem, cudaMemcpyHostToDevice);
  dim3 block = {256, 1, 1};
  dim3 grid = {(nelem / block.x) + (nelem % block.x ? 1 : 0), 1, 1};
  vecAddKernel<<<grid, block>>>(A, B, C, nelem);
  cudaMemcpy(C_h, C, sizeof(float) * nelem, cudaMemcpyDeviceToHost);
  cudaFree(dA);
  cudaFree(dB);
  cudaFree(dC);
}
```



GPU

Not Portable

OR

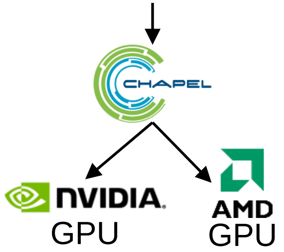
Device
Allocate /
Free

How we get to talk
about today =)

Chapel

```
use GPU;
proc vecAdd(A_h: [] real(32),
  B_h: [] real(32), C_h: [] real(32)) {
  on here.gpus[0] { //GPU locale
    var A: [A_h.domain] real(32);
    var B: [B_h.domain] real(32);
    var C: [C_h.domain] real(32);
    A = A_h; //copy-to
    B = B_h; //copy-to
    C = A + B; //GPU add
    C_h = C; //copy-from
  }
}
```

(implicit)



GPU | GPU

Portable

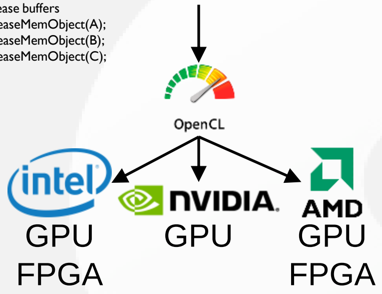
Least Programmable — Most Programmable

How would you rather add vectors on a GPU?

How I learned and
spent 10+ years

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) {
    C[tid] = A[tid] + B[tid];
  }
}
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) * work, 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) * work, C_h, 0, NULL, NULL);
  clFinish(q);
  //Release buffers
  clReleaseMemObject(A);
  clReleaseMemObject(B);
  clReleaseMemObject(C);
}
```



Portable

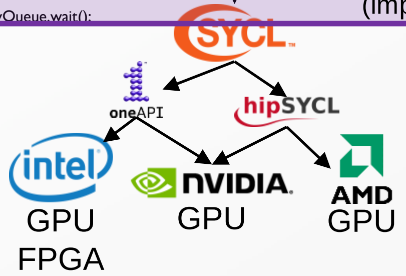
How I've been doing
it recently (on FPGA)

SYCL

```
void vecAdd(float *A_h, float *B_h, float *C_h, int32_t nelem) {
  sycl::queue myQueue;
  sycl::buffer<float> A(A_h, nelem, sycl::property::buffer::use_host_ptr());
  sycl::buffer<float> B(B_h, nelem, sycl::property::buffer::use_host_ptr());
  sycl::buffer<float> C(C_h, nelem, sycl::property::buffer::use_host_ptr());
  C.set_write_back(true);
  sycl::range<1> local{256};
  sycl::range<1> global{((nelem / local.get(0)) + (nelem % local.get(0) ? 1 : 0)) * local.get(0)};
  myQueue.submit([&](sycl::handler &cgh) { //GPU submit
    auto A_acc = A.get_access<sycl::access::mode::read>(cgh, sycl::range<1>{(size_t)nelem});
    auto B_acc = B.get_access<sycl::access::mode::read>(cgh, sycl::range<1>{(size_t)nelem});
    auto C_acc = C.get_access<sycl::access::mode::discard_write>(cgh, sycl::range<1>{(size_t)nelem});
    cgh.parallel_for(sycl::nd_range<1>{global, local}, [=](sycl::nd_item<1> tid_info) {
      size_t tid = tid_info.get_global_linear_id();
      if (tid < nelem) {
        C_acc[tid] = A_acc[tid] + B_acc[tid];
      }
    });
  });
  myQueue.wait();
}
```

("when" is implicit)

(implicit)

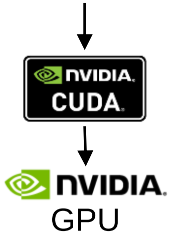


Portable

How **most** GPU
kernels are written

CUDA

```
__global__ void vecAddKernel(float *A, float *B, float *C, int32_t nelem) {
  size_t tid = blockDim.x * blockIdx.x + threadIdx.x;
  if (tid < nelem) {
    C[tid] = A[tid] + B[tid];
  }
}
void vecAdd(float *A_h, float *B_h, float *C_h, int32_t nelem) {
  float *A, *B, *C;
  int32_t work = hi-lo+1;
  cudaMalloc(&A, sizeof(float) * nelem);
  cudaMalloc(&B, sizeof(float) * nelem);
  cudaMalloc(&C, sizeof(float) * nelem);
  cudaMemcpy(A, A_h, sizeof(float) * nelem, cudaMemcpyHostToDevice);
  cudaMemcpy(B, B_h, sizeof(float) * nelem, cudaMemcpyHostToDevice);
  dim3 block = {256, 1, 1};
  dim3 grid = {(nelem / block.x) + (nelem % block.x ? 1 : 0), 1, 1};
  vecAddKernel<<<grid, block>>>(A, B, C, nelem);
  cudaMemcpy(C_h, C, sizeof(float) * nelem, cudaMemcpyDeviceToHost);
  cudaFree(dA);
  cudaFree(dB);
  cudaFree(dC);
}
```



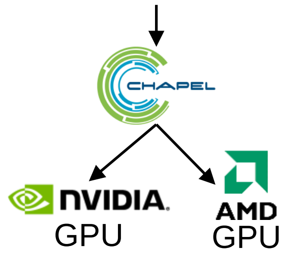
Not Portable

OR

How we get to talk
about today =)

Chapel

```
use GPU;
proc vecAdd(A_h: [] real(32),
            B_h: [] real(32), C_h: [] real(32)) {
  on here.gpus[0] { //GPU locale
    var A: [A_h.domain] real(32);
    var B: [B_h.domain] real(32);
    var C: [C_h.domain] real(32);
    A = A_h; //copy-to
    B = B_h; //copy-to
    C = A + B; //GPU add
    C_h = C; //copy-from
  }
}
```



Portable

X-fer
CPU→GPU
GPU→CPU

Least Programmable

Most Programmable

Why *Graph Analysis on* GPU via Chapel?

Understand Chapel's *programmability* and *GPU performance* on *irregular applications* relative to proven approaches

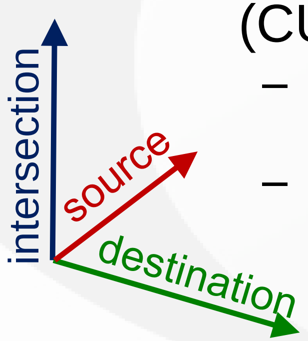
Graph Workload: Edge-connected Jaccard similarity

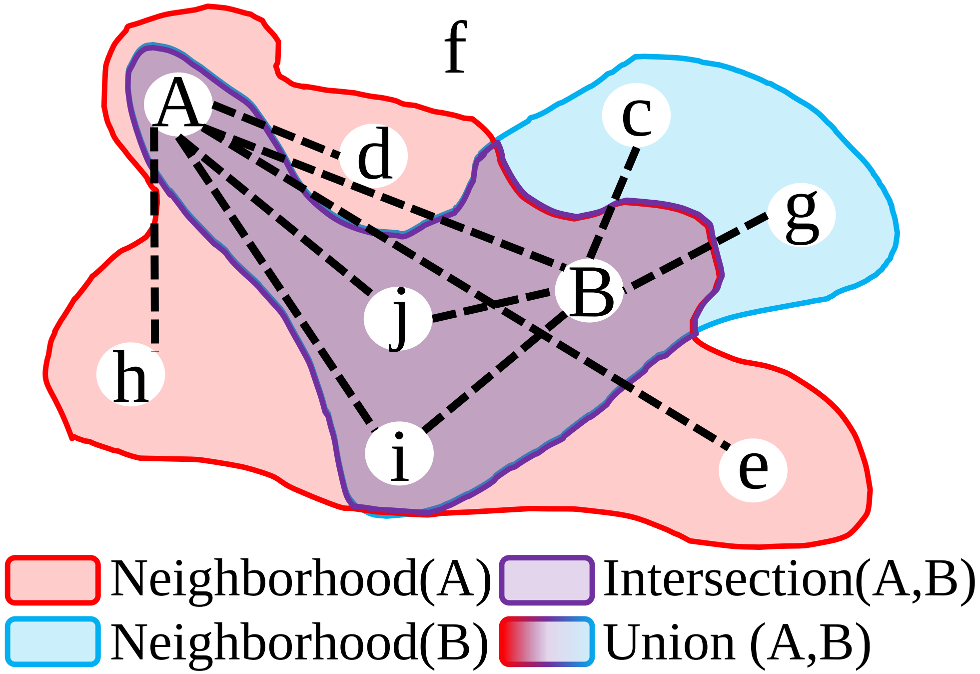
- Intersection over union for all edge-connected pairs' 1-hop neighborhoods
  - Essentially a batch of |E| set-intersections
  - Lots of indirect access

- 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

A. Fender, et al, 2017. "Parallel Jaccard and Related Graph Clustering Techniques." *(ScalA '17).*





$$\text{Jaccard Similarity}(A,B) = \frac{\text{Intersection}(A,B)}{\text{Union }(A,B)}$$

```
Edges = {(a,b), (a,d), (a,e), (a,h), (a,i),
         (a,j), (b,c), (b,g), (b,i), (b,j)}
forall e in Edges {
{
    JS(e) = intersect(e)/union(e)
}
```

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

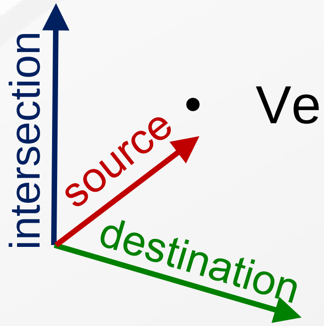
- C-style Chapel: 756 lines
  - Edge-centric: 150
  - Vertex-Centric: 230
  - Everything else: 376

- CUDA: 1212
  - All Kernels: 395
  - Everything else: 817

```
template <typename vertex_t, typename
__global__ void jaccard_ec (vertex_t
*weight_j) {
  ... //thread-private vars
  tid = blockIdx.x * blockDim.x + thr
  if (tid < e) {
    ... // choose smaller vert as ‘re
    for (i = csrPtr[ref]; i < csrPtr[
      ref_col = csrInd[i];
      ... // bin-search for ref_col b
      ... // accumulate matches in we
    }
    ...
  }
}
```

```
forall i in indices.domain {
    assertOnGpu(); //Fail if this can
    ... // thread-private vars
    ... // choose smaller vert as ‘re
    for j in offsets[refs]..<(offsets
      ref_col = indices[j];
      ... // bin-search for ref_col b
      ... // accumulate matches in we
    }
  }
```

Vertex-Centric: More complex



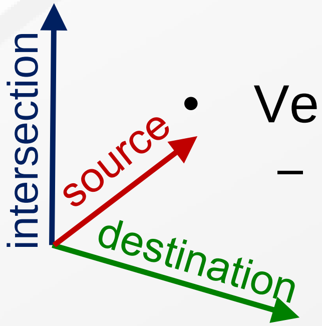
- Vertex-centric intersect didn't initially GPU-ize

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

VC-CPU-Naïve

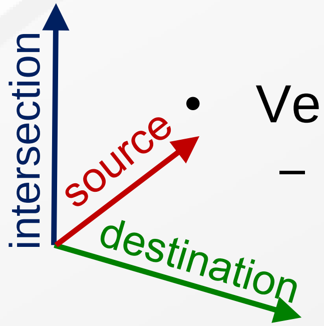
```
var intersect : [0..<numEdges] atomic real(32);
forall Z in srcIters by gridDim.z*blockDim.z {
  forall Y in destIters by gridDim.y*blockDim.y {
    forall X in isectIters by gridDim.x*blockDim.x {
      ... // bin-search
      intersect[writeAddr].add(1.0);
    }
  }
}
```

VC-CPU-Linearized

```
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; }
}
```

(↑ 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-Naïve

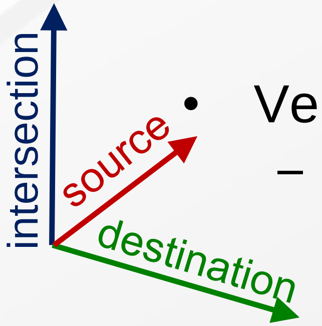
```
var intersect : [0..<numEdges] atomic real(32);
forall Z in srcIters by gridDim.z*blockDim.z {
  forall Y in destIters by gridDim.y*blockDim.y {
    forall X in isectIters by gridDim.x*blockDim.x {
      ... // bin-search
      intersect[writeAddr].add(1.0);
    }
  }
}
```

VC-CPU-Linearized

```
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; }
}
```

(↑ 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

```
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

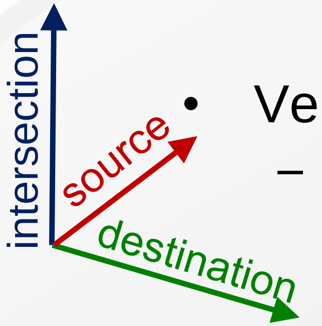
```
var intersect : [0..<numEdges] atomic real(32);
forall id in srcIters*destIters*isectIters {
  var nd_id : 3*int = get_ND_ID(id);
  var zCount = nd_id(2);
  while (zCount < zMax) {
    var yCount = nd_id(1);
    while (yCount < yMax) {
      var xCount = nd_id(0);
      while (xCount < xMax) {
        ... // 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

```
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

```
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)
```

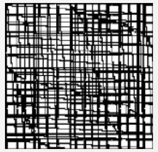
(↑ pseudo-code, see Github for real)

How well did it perform?

Test Data

Sparsest to densest

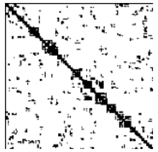
Kmer_A2a
Protein *k*-mers
V: 171M
E: 361M
Degree
Avg: 2.11
Range: 39
Std. Dev.: 0.56
Gini Index: 0.055



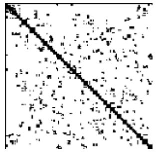
Europe_osm
European roads
V: 50.9M
E: 108M
Degree
Avg: 2.12
Range: 12
Std. Dev.: 0.48
Gini Index: 0.085

Road_usa
US roads
V: 23.9M
E: 57.7M
Degree
Avg: 2.41
Range: 8
Std. Dev.: 0.93
Gini Index: 0.211

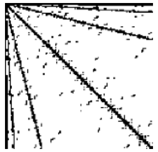
Road-roadNet-CA
California roads
V: 1.96M
E: 5.52M
Degree
Avg: 2.82
Range: 11
Std. Dev.: 0.99
Gini Index: 0.185



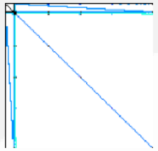
Road-roadNet-PA
Pennsylvania roads
V: 1.09M
E: 3.08M
Degree
Avg: 2.83
Range: 8
Std. Dev.: 1.02
Gini Index: 0.188



Delaunay_n24
Random Triangulations
V: 16.8M
E: 101M
Degree
Avg: 6.00
Range: 23
Std. Dev.: 1.34
Gini Index: 0.122



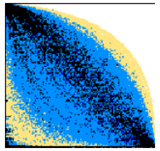
circuit5M
Large circuit
V: 5.56M
E: 54.0M
Degree
Avg: 9.71
Range: 1.29M
Std. Dev.: 1357
Gini Index: 0.577



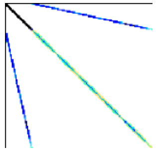
Soc-LiveJournal1
Social network
V: 4.85M
E: 85.7M
Degree
Avg: 17.7
Range: 20.3K
Std. Dev.: 52.0
Gini Index: 0.711

Wikipedia-20070206
Web page links
V: 3.57M
E: 84.8M
Degree
Avg: 23.8
Range: 188K
Std. Dev.: 255
Gini Index: 0.759

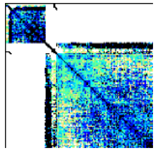
GL7d19
Voroni differentials
V: 1.96M
E: 74.6M
Degree
Avg: 38.2
Range: 134
Std. Dev.: 6.73
Gini Index: 0.088



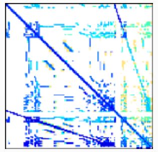
dielFilterV2real
Dielectric resonator
V: 1.16M
E: 47.4M
Degree
Avg: 40.9
Range: 104
Std. Dev.: 16.1
Gini Index: 0.201



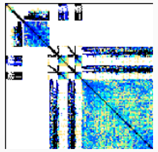
Sc-ldoor
Large door
V: 952K
E: 41.5M
Degree
Avg: 43.6
Range: 76
Std. Dev.: 14.8
Gini Index: 0.183



Stokes
VLSI process sim.
V: 11.4M
E: 516M
Degree
Avg: 45.1
Range: 1728
Std. Dev.: 61.8
Gini Index: 0.392



Sc-msdoor
Medium Door
V: 416K
E: 18.8M
Degree
Avg: 45.1
Range: 76
Std. Dev.: 13.7
Gini Index: 0.166

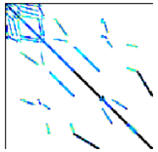


Ca-coauthors-dblp
Coauthorship
V: 540K
E: 30.5M
Degree
Avg: 56.4
Range: 3298
Std. Dev.: 66.2
Gini Index: 0.544

Soc-orkut
Social network
V: 3.00M
E: 213M
Degree
Avg: 71.0
Range: 27.5K
Std. Dev.: 140
Gini Index: 0.558

Hollywood-2009
Costarring Actors
V: 1.14M
E: 113M
Degree
Avg: 98.9
Range: 11.5K
Std. Dev.: 272
Gini Index: 0.750

HV15R
CFD of engine fan
V: 2.02M
E: 325M
Degree
Avg: 161
Range: 491
Std. Dev.: 47.8
Gini Index: 0.155

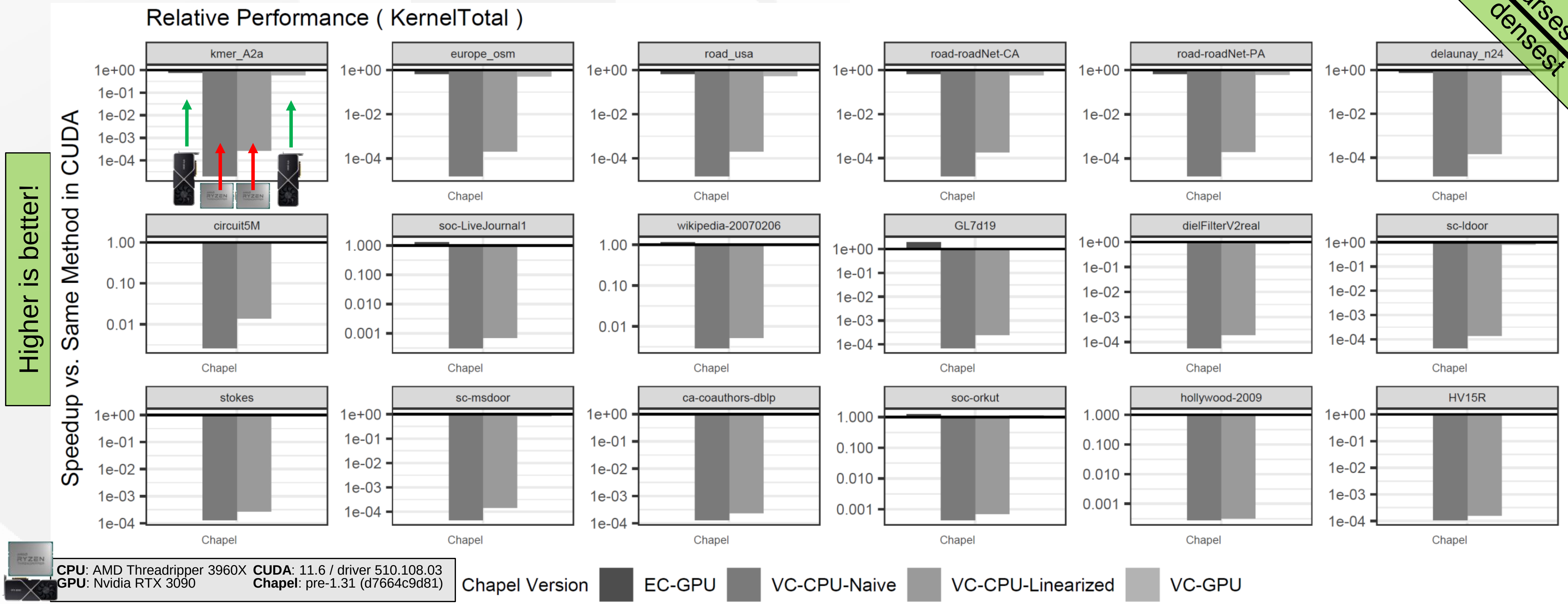


Graph data and images CC-BY-4.0 from the SparseSuite Matrix Collection (https://sparse.tamu.edu/).

Preprocessed CSR binary files: https://chrec.cs.vt.edu/SYCL-Jaccard/HPEC22-Data/index.html

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



Higher is better!

Sparsest to densest

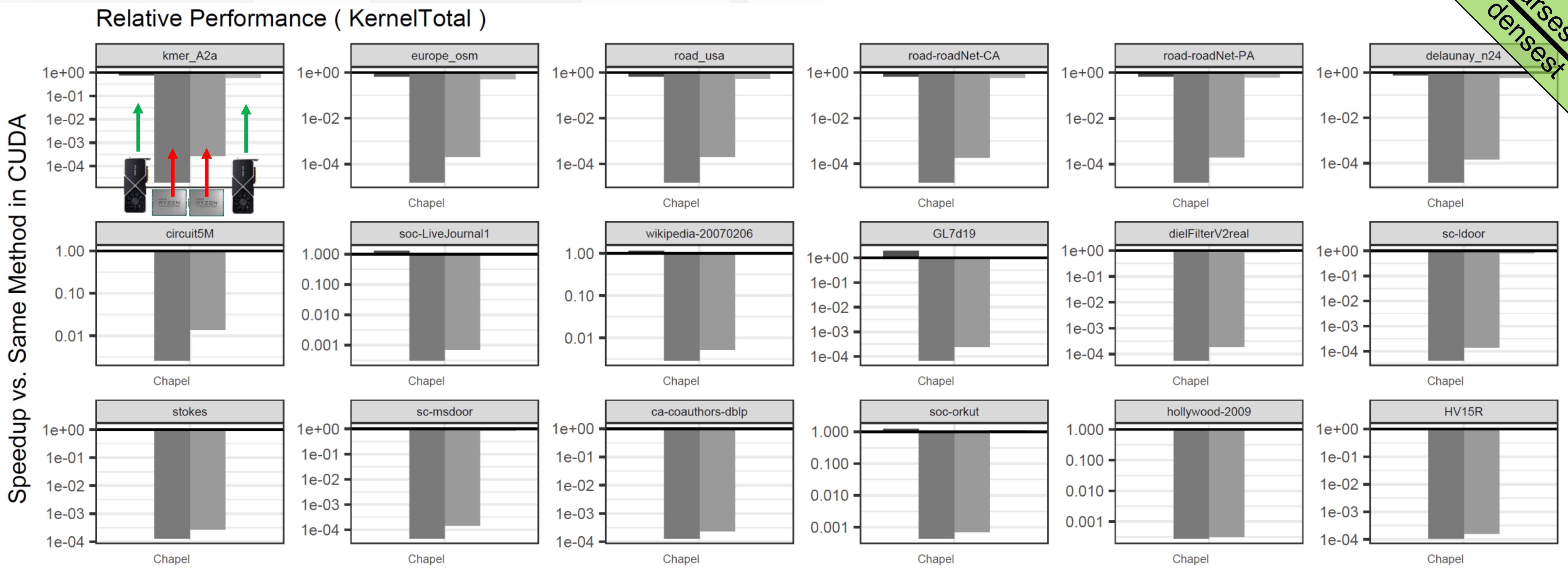
CPU: AMD Threadripper 3960X **CUDA**: 11.6 / driver 510.108.03
GPU: Nvidia RTX 3090 **Chapel**: pre-1.31 (d7664c9d81)

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...

Sparsest to densest

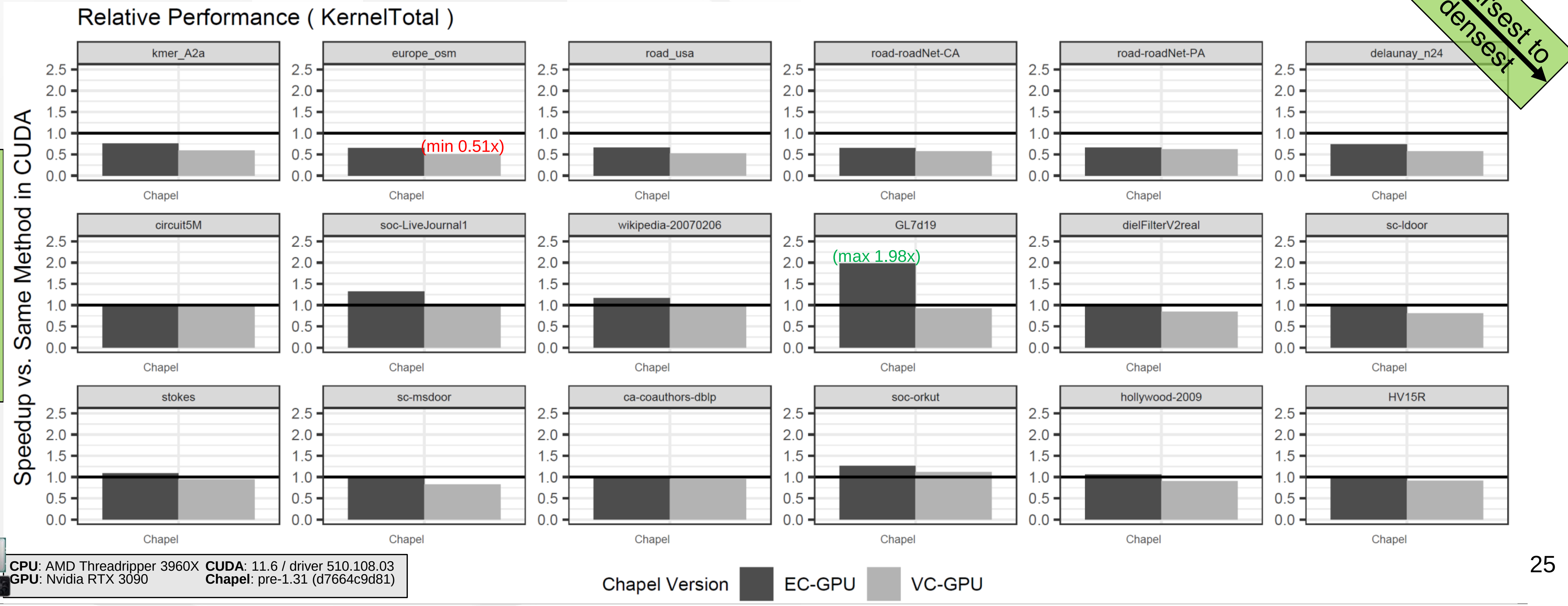
Higher is better!



Chapel Version ■ EC-GPU ■ VC-CPU-Naive ■ VC-CPU-Linearized ■ VC-GPU

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

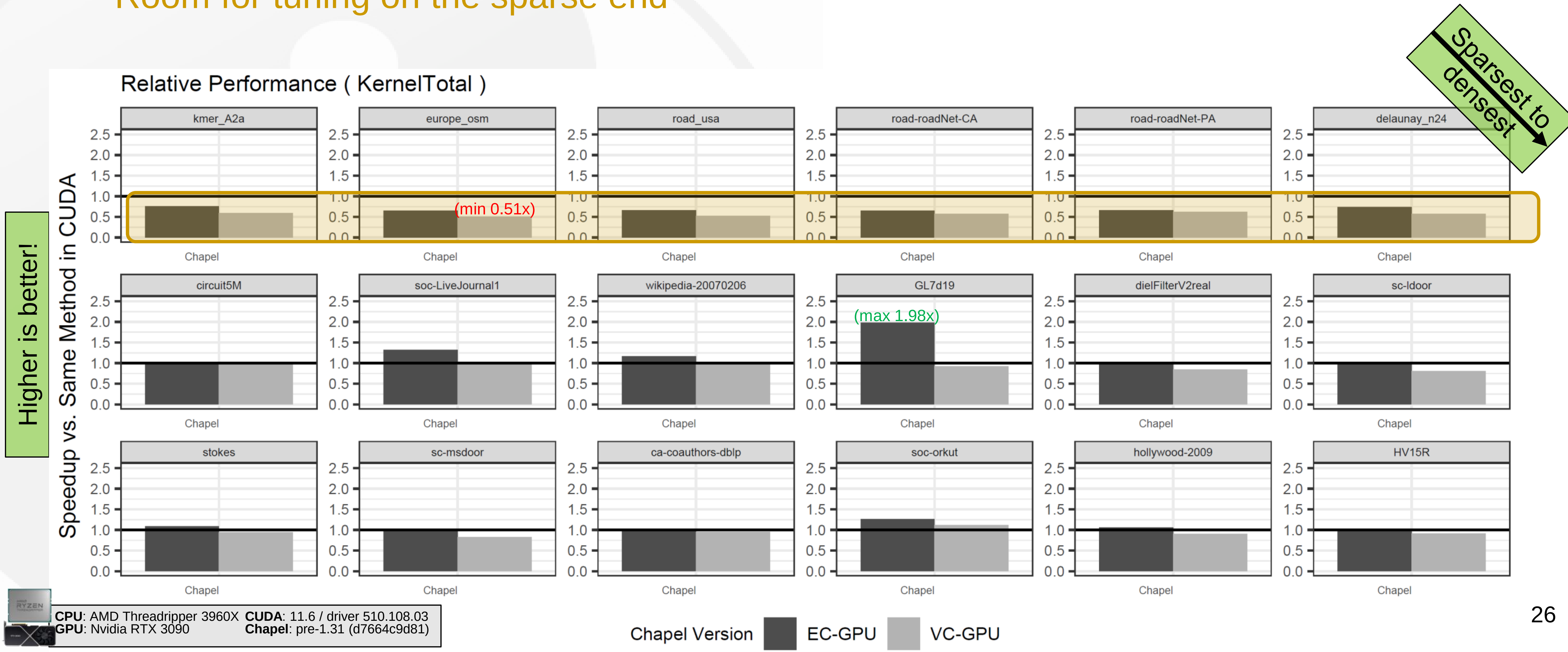
GPU Performance is pretty good right out of the box



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

GPU Performance is pretty good right out of the box

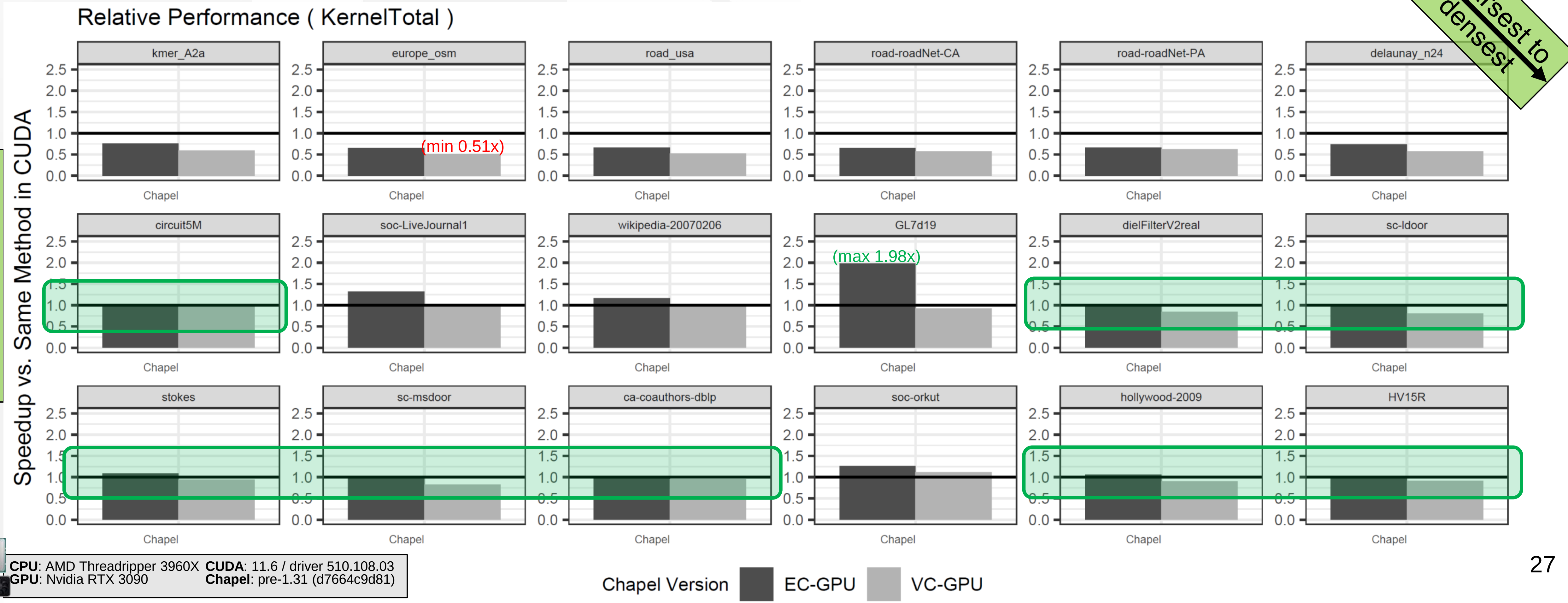
- Room for tuning on the sparse end



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

GPU Performance is pretty good right out of the box

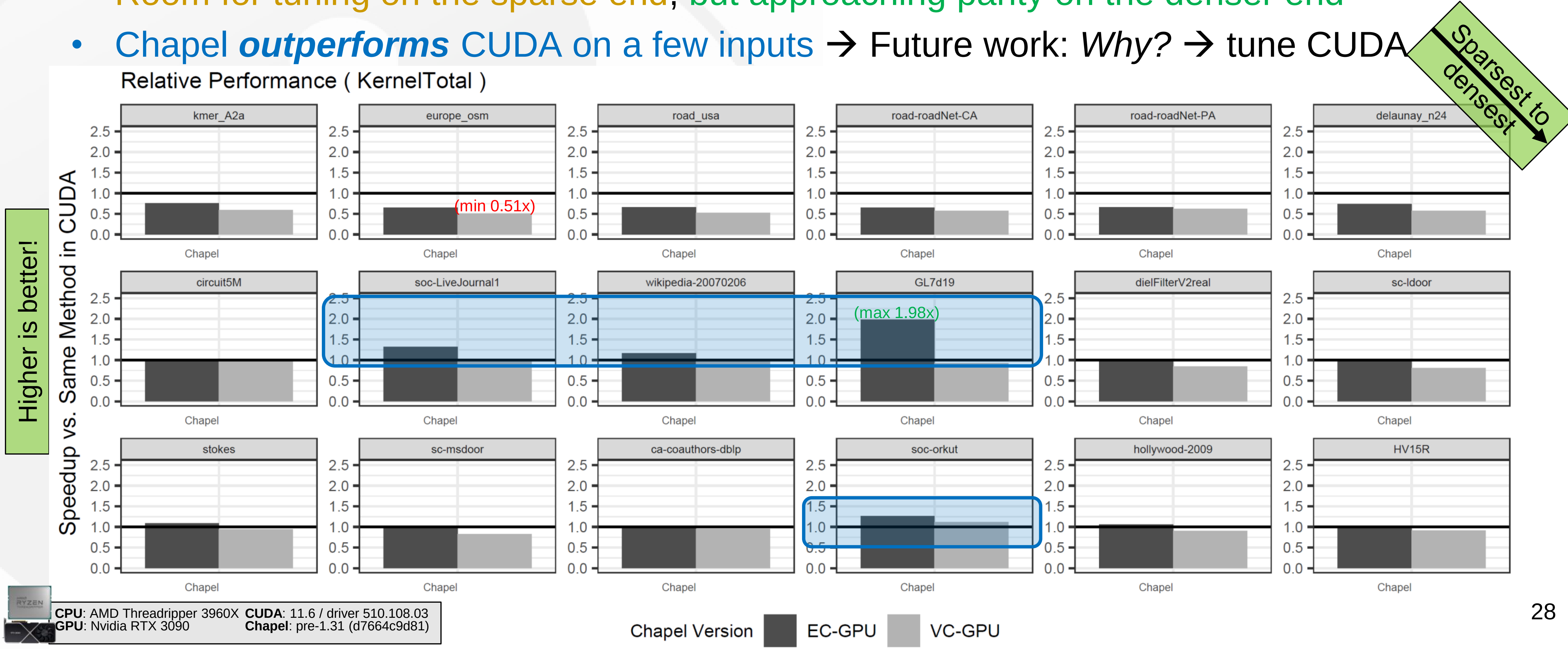
- Room for tuning on the sparse end, but approaching parity on the denser end



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

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



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

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

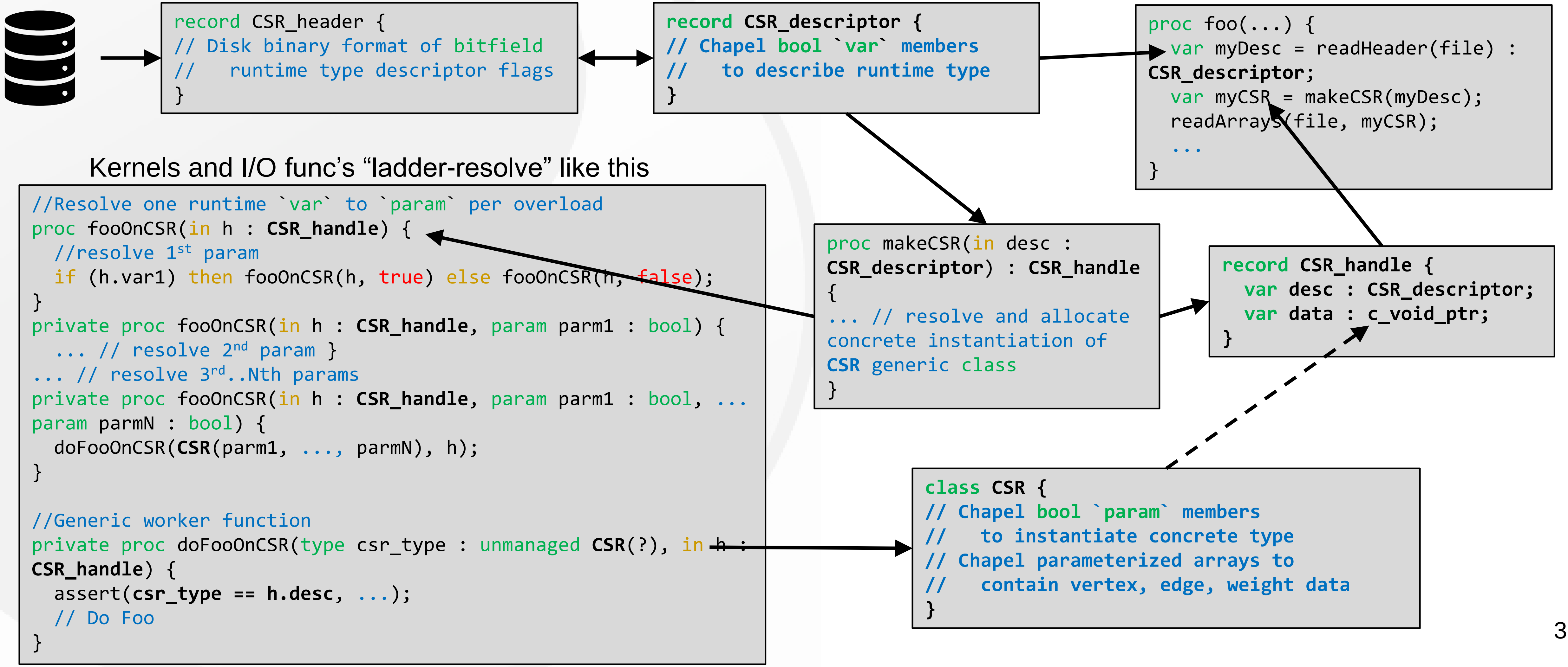
GitHub
Code: https://github.com/vtsynergy/Chapel-Examples
Input data: https://chrec.cs.vt.edu/SYCL-Jaccard/HPEC22-Data/index.html

Mission-Critical Computing
NSF CENTER FOR SPACE, HIGH-PERFORMANCE, AND RESILIENT COMPUTING (SHREC)

The work detailed herein has been supported in part by NSF I/UCRC CNS-1822080 via the NSF Center for Space, High-performance, and Resilient Computing (SHREC).

Backup Slides

Rudimentary *Explicit* RTTI: How we did it (C-style)



```
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);
  ...
}
```

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

```
record CSR_handle {
  var desc : CSR_descriptor;
  var data : c_void_ptr;
}
```

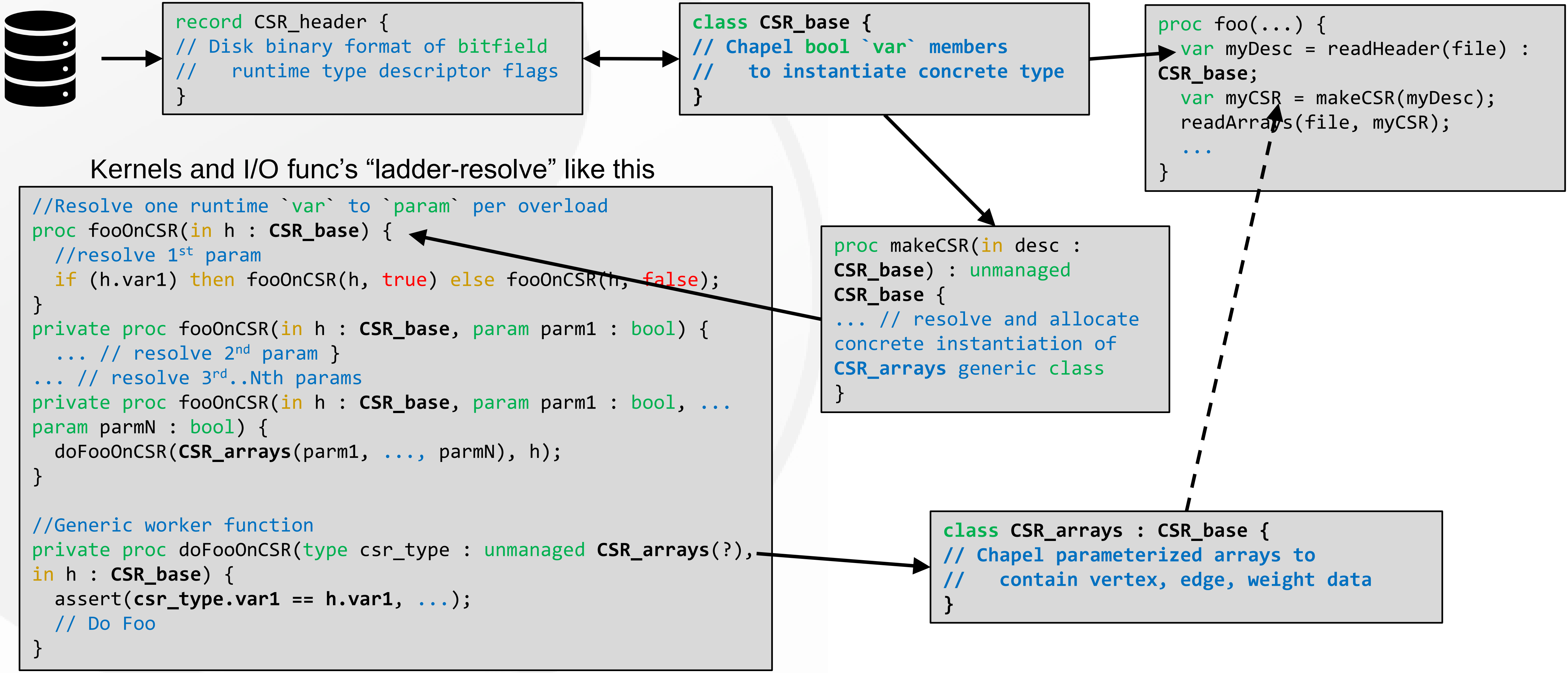
Kernels and I/O func's "ladder-resolve" like this

```
//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
}
```

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

Rudimentary RTTI: How we'd redo it (Chapel/OO-style)



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

```
class CSR_base {
// Chapel bool `var` members
//   to instantiate concrete type
}
```

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

Kernels and I/O func's "ladder-resolve" like this

```
//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
}
```

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

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