Kokkos: Core Libraries

being developed @ Sandia, LLC. (federally funded) since 2014

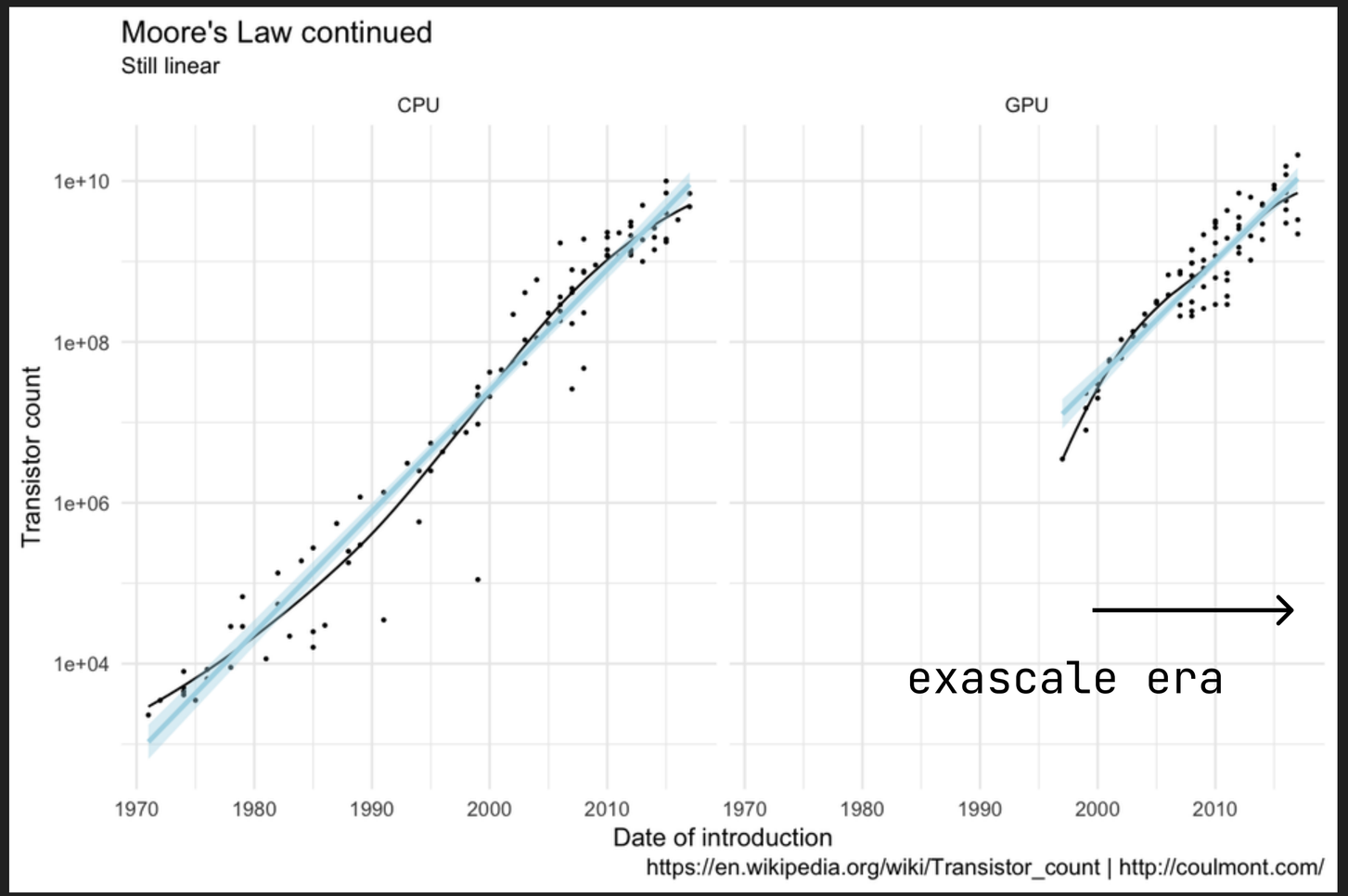
C++ framework for writing performance portable applications targeting all major HPC platforms

Hayk Hakobyan (PPPL/Columbia)

all the slides + examples and code samples available: https://github.com/haykh/princeton-bootcamp-kokkos-2022

Why would I care about GPUs?

The Moore’s Law is dead,
Long live the Moore’s Law!



Aurora will use Intel GPUs

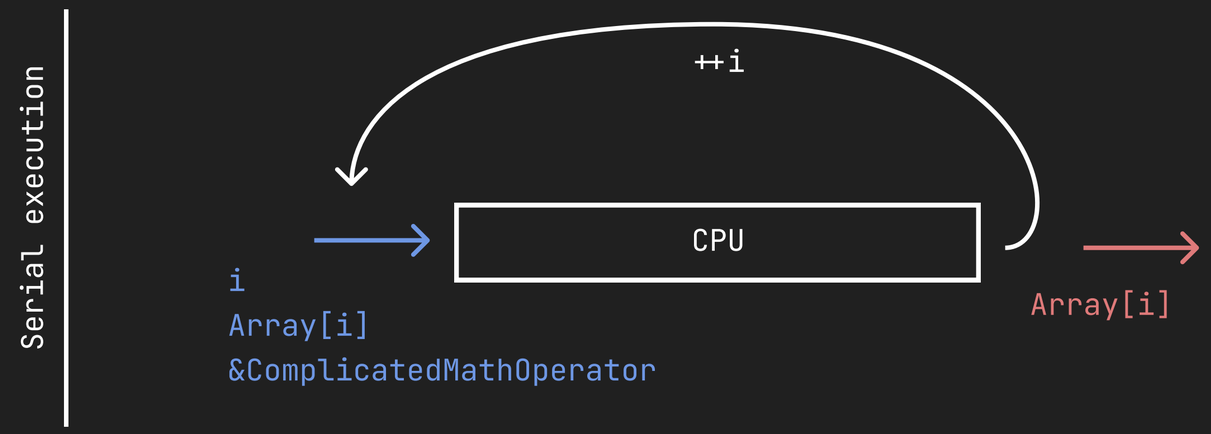
Summit uses NVIDIA GPUs

image credits: @bjco from reddit.com/r/dataisbeautiful

+ more energy efficient (high FLOPS/GW)

CPU (serial) vs GPU (vectorized) programming[*]

- Execution policy

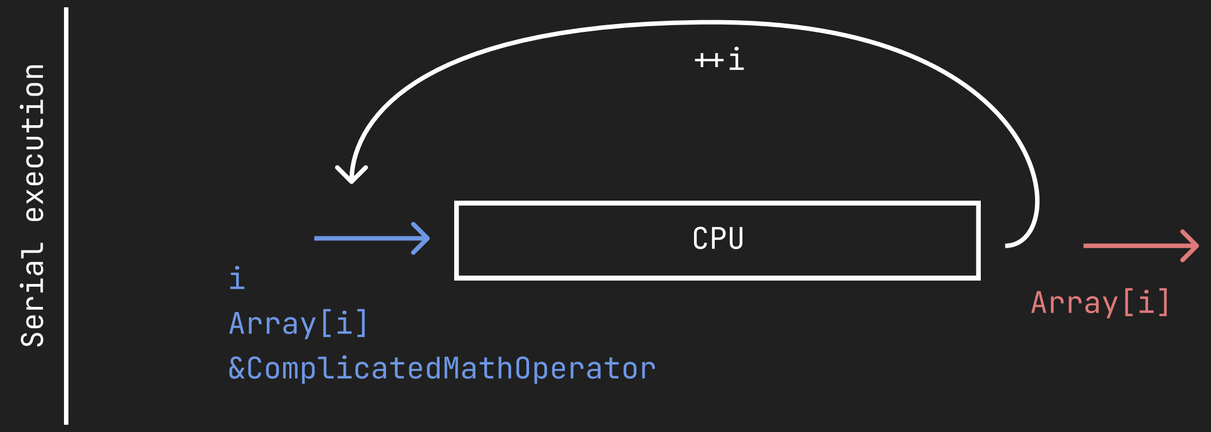



```
for (i = 0; i < GAZILLION; ++i) {
  Array[i] = ComplicatedMathOperation(i);
}
```

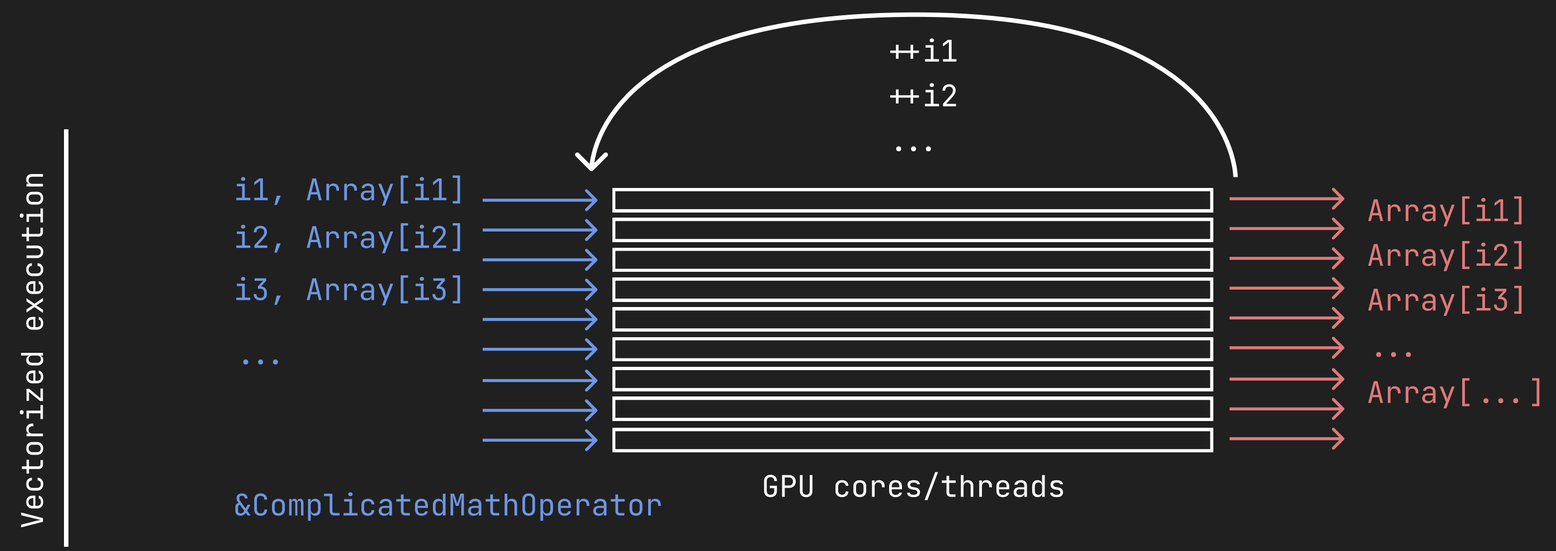
[*] violently oversimplified

CPU (serial) vs GPU (vectorized) programming[*]

- Execution policy



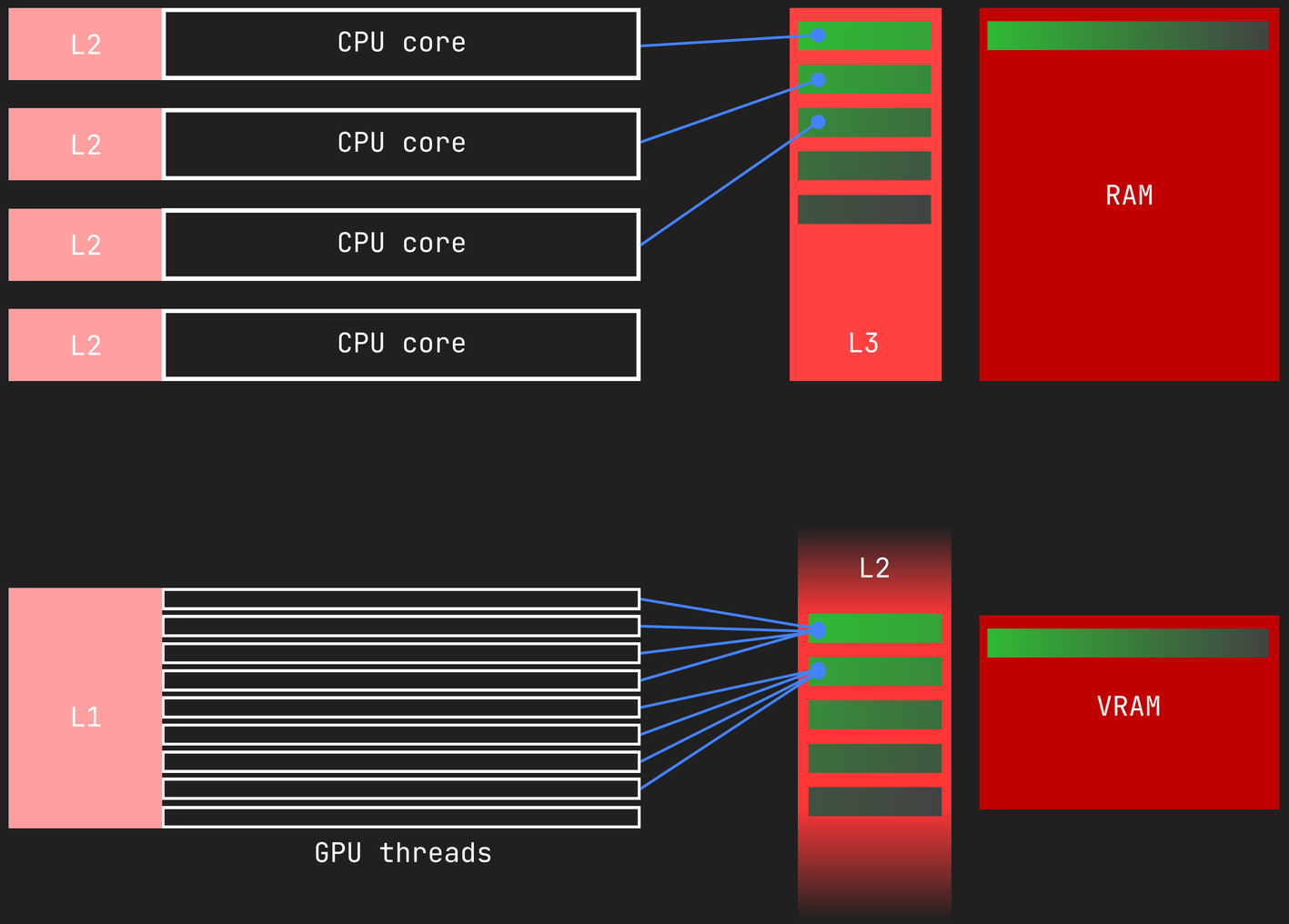

```
for (i = 0; i < GAZILLION; ++i) {
  Array[i] = ComplicatedMathOperation(i);
}
```



[*] violently oversimplified

CPU (serial) vs GPU (vectorized) programming[*]

- Memory layout

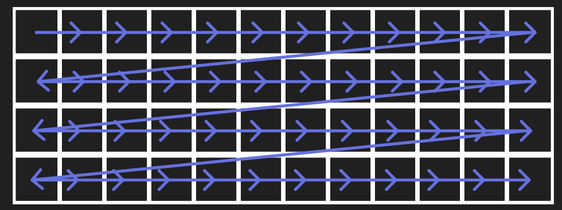


row-major

cached memory access

good on CPUs

bad on GPUs

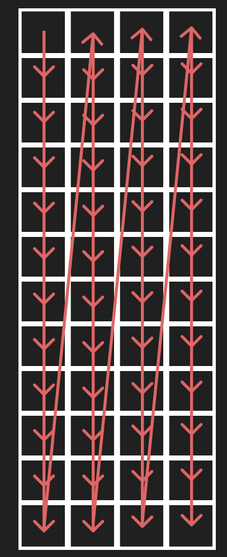


column-major

coalesced memory access

bad on CPUs

good on GPUs



[*] violently oversimplified

CPU (serial) vs GPU (vectorized) programming[*]

When writing a CPU/GPU portable code,
"one size fits all" is inefficient

- execution sequence becomes important
  - serial *vs* vectorized
  - ensure no race conditions
  - ensure no data dependency
- memory access type should depend on the architecture
  - minimize data movement on GPUs
  - computing on-the-fly *vs* tabulating
  - coalesced access *vs* caching
  - optimal sizes of data chunks (blocks, tiles) may vary

[*] violently oversimplified

Philosophy behind Kokkos

```
#include <complex.h>
#include <stdio.h>
#include <time.h>
int *d, D[9999], N=20, L=4, n, m, k, a[3], i;char *p, *q, S[2000]="L@X-SGD-HNBBB-AD-VHSG--XNT\x1b[2J\x1b[H",
*s=S, *G="r2zZX!.+@KBK^yh.!:%Bud!.+Jyh.!6.BHBh!6.BHBh!:%Buv{VT!.hBJ6p!042ljn!284b}`!.hR6Dp!.hp!h.T6p!h.p6!
2/LilqP72!h.+@QB!~}lqP72/Lil!h.+@QBFp!:)0?F]nwf!,82v!.sv{6!.l6!,j<n8!.xN6t!&NvN*!.6hp";
/*nqncgrqsebzlhfhxrraqbujjvggre:@znzrggre*/typedef complex double(c);c(X)[3], P, O;c B(double t) {double s=1-
t, u;P=s*s*X[1]+2*s*t**X+t*t*X[2]+O;u=I*P;return +48*((s=P)+48*I)/(1<u?u:1);}void b(double t, double u)
{double s=P=B(t)-B(u);(s=P*(2*s-P))<1?m=P=B((t+u)/2), k=-I*P,m > -41 &&m<39 &&9< k &&k<48?
m+=k/2*80+73,S[m]=S[m]-73?k %2? S[m]-94?95:73:S[m]-95?94:73:73: 1: (b(t, (t+u)/2), b((t+u)/2, u), 0);}int
main(int(x), char **V) {int aug=0, augn=59;const char
gammds[]="ARIRE\nTBAAN\nTVIR\nLBH\nHC";clock_t(c)=clock();for (d=D; m<26; m++, s++)*s > 63?*d++=m %7*
16-7*8, *d++=m/7*25, *d++=*s-64:0;for (d=D, L=N=m=0; aug<augn; aug++) {x=gammds[aug];if (x > 13?64<x &&x<91?
*d++=m*16, *d++=L*25,*d++=x % 26:0, m++,0: 1)for (++L; d-D > N*3||(m=0); N++)D[N*3] -= m*8;}for (;
i<200+L*25; i++) {for (n=0, p=S+33; n<1920; *p++=n++ % 80> 78?10:32) {}for (*p=x=0, d=D; x<N; x++, d+=3)
{O=(d[1]-i-40)*I+*d;n=d[2];p=G;for (; n--;) {for (; *p++ > 33;) {}}*a=a[1]=*p++;for (; *p > 33; p++)if (*p
%2? *a=*p,0: 1) {a[2]=*p;for (m=0; m<3; m++) {k=a[m]/2-18;q="/&&&##%%##.+),A$$$$'&&'&&((%-((#'/#%
%#&#\&&#D&";for (n=2; k--;)n+=*q++-34;X[m]=n % 13+n/13*I;}b(0, 1);*a=a[1]=*p;}}for (puts(s), s=S+30;
(clock()-c)*20<i*CLOCKS_PER_SEC;) {}}return 0;}
```

as scientists we want to focus
on writing
science code
not housekeeping or porting

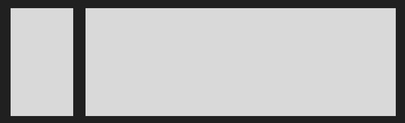
Philosophy behind Kokkos

```
#include <complex.h>
#include <stdio.h>
#include <time.h>
int *d, D[9999], N=20, L=4, n, m, k, a[3], i;char *p, *q, S[2000]="L@X-SGD-HNBBB-AD-VHSG--XNT\x1b[2J\x1b[H",
*s=S, *G="r2zZX!.+@KBK^yh.!:%Bud!.+Jyh.!6.BHBhp!6.BHBh!:%Buv{VT!.hBJ6p!042ljn!284b}`!.hR6Dp!.hp!h.T6p!h.p6!
2/LilqP72!h.+@QB!~}lqP72/Lil!h.+@QBFp!:)0?F]nwf!,82v!.sv{6!.l6!,j<n8!.xN6t!&NvN*!.6hp";
/*nqncgrqsebzlhfhxrraqbujjvggre:@znzrggre*/typedef complex double(c);c(X)[3], P, O;c B(double t) {double s=1-
t, u;P=s*s*X[1]+2*s*t**X+t*t*X[2]+O;u=I*P;return +48*((s=P)+48*I)/(1<u?u:1);}void b(double t, double u)
{double s=P=B(t)-B(u);(s=P*(2*s-P))<1?m=P=B((t+u)/2), k=-I*P,m > -41 &&m<39 &&9< k &&k<48?
m+=k/2*80+73,S[m]=S[m]-73?k %2? S[m]-94?95:73:S[m]-95?94:73:73: 1: (b(t, (t+u)/2), b((t+u)/2, u), 0);}int
main(int(x), char **V) {int aug=0, augn=59;const char
gammds[]="ARIRE\nTBAAN\nTVIR\nLBH\nHC";clock_t(c)=clock();for (d=D; m<26; m++, s++)*s > 63?*d++=m %7*
16-7*8, *d++=m/7*25, *d++=*s-64:0;for (d=D, L=N=m=0; aug<augn; aug++) {x=gammds[aug];if (x > 13?64<x &&x<91?
*d++=m*16, *d++=L*25,*d++=x % 26:0, m++,0: 1)for (++L; d-D > N*3||(m=0); N++)D[N*3] -= m*8;}for (;
i<200+L*25; i++) {for (n=0, p=S+33; n<1920; *p++=n++ % 80> 78?10:32) {}for (*p=x=0, d=D; x<N; x++, d+=3)
{O=(d[1]-i-40)*I+*d;n=d[2];p=G;for (; n--;) {for (; *p++ > 33;) {}}*a=a[1]=*p++;for (; *p > 33; p++)if (*p
%2? *a=*p,0: 1) {a[2]=*p;for (m=0; m<3; m++) {k=a[m]/2-18;q="/&&&##%%##.+),A$$$$'&&'&&((%-((#'/#%
%#&#\&&#D&";for (n=2; k--;)n+=*q++-34;X[m]=n % 13+n/13*I;}b(0, 1);*a=a[1]=*p;}}for (puts(s), s=S+30;
(clock()-c)*20<i*CLOCKS_PER_SEC;) {}}return 0;}
```

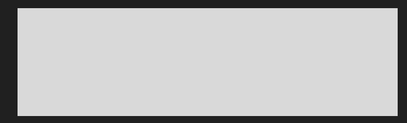
as scientists we want to focus
on writing
science code
not housekeeping or porting

one code that compiles seamlessly with
different compilers/architectures *and*
runs efficiently

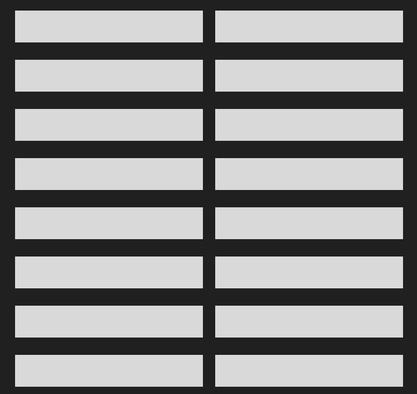
PC with a GPU
(e.g., nvcc + gcc)



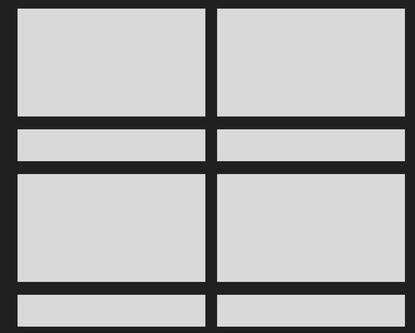
Personal laptop
(e.g., llvm compiler)



CPU cluster
(e.g., AOCC compiler)



CPU/GPU cluster
(e.g., nvc + icc compilers)



Philosophy behind Kokkos

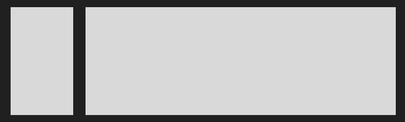
```
#include <complex.h>
#include <stdio.h>
#include <time.h>
int *d, D[9999], N=20, L=4, n, m, k, a[3], i;char *p, *q, S[2000]="L@X-SGD-HNBBB-AD-VHSG--XNT\x1b[2J\x1b[H",
*s=S, *G="r2zZX!.+@KBK^yh.!:%Bud!.+Jyh.!6.BHBhp!6.BHBh!:%Buv{VT!.hBJ6p!042ljn!284b}`!.hR6Dp!.hp!h.T6p!h.p6!
2/LilqP72!h.+@QB!~}lqP72/Lil!h.+@QBFp!:)0?F]nwf!,82v!.sv{6!.l6!,j<n8!.xN6t!&NvN*!.6hp";
/*nqncgrqsebzlhfhxrraqbujjvggre:@znzrggre*/typedef complex double(c);c(X)[3], P, O;c B(double t) {double s=1-
t, u;P=s*s*X[1]+2*s*t**X+t*t*X[2]+O;u=I*P;return +48*((s=P)+48*I)/(1<u?u:1);}void b(double t, double u)
{double s=P=B(t)-B(u);(s=P*(2*s-P))<1?m=P=B((t+u)/2), k=-I*P,m > -41 &&m<39 &&9< k &&k<48?
m+=k/2*80+73,S[m]=S[m]-73?k %2? S[m]-94?95:73:S[m]-95?94:73:73: 1: (b(t, (t+u)/2), b((t+u)/2, u), 0);}int
main(int(x), char **V) {int aug=0, augn=59;const char
gammds[]="ARIRE\nTBAAN\nTVIR\nLBH\nHC";clock_t(c)=clock();for (d=D; m<26; m++, s++)*s > 63?*d++=m %7*
16-7*8, *d++=m/7*25, *d++=*s-64:0;for (d=D, L=N=m=0; aug<augn; aug++) {x=gammds[aug];if (x > 13?64<x &&x<91?
*d++=m*16, *d++=L*25,*d++=x % 26:0, m++,0: 1)for (++L; d-D > N*3||(m=0); N++)D[N*3] -= m*8;}for (;
i<200+L*25; i++) {for (n=0, p=S+33; n<1920; *p++=n++ % 80> 78?10:32) {}for (*p=x=0, d=D; x<N; x++, d+=3)
{O=(d[1]-i-40)*I+*d;n=d[2];p=G;for (; n--;) {for (; *p++ > 33;) {}}*a=a[1]=*p++;for (; *p > 33; p++)if (*p
%2? *a=*p,0: 1) {a[2]=*p;for (m=0; m<3; m++) {k=a[m]/2-18;q="/&&&##%%##.+),A$$$$'&&'&&((%-((#'/#%
%#&#\&&#D&";for (n=2; k--;)n+=*q++-34;X[m]=n % 13+n/13*I;}b(0, 1);*a=a[1]=*p;}}for (puts(s), s=S+30;
(clock()-c)*20<i*CLOCKS_PER_SEC;) {}}return 0;}
```

Kokkos

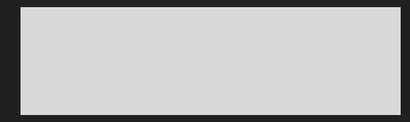
kokkos::View
kokkos::RangePolicy
kokkos::parallel_for
...

as scientists we want to focus
on writing
science code
not housekeeping or porting

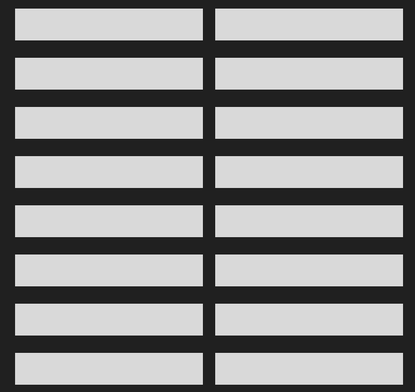
PC with a GPU
(e.g., nvcc + gcc)



Personal laptop
(e.g., llvm compiler)



CPU cluster
(e.g., AOCC compiler)



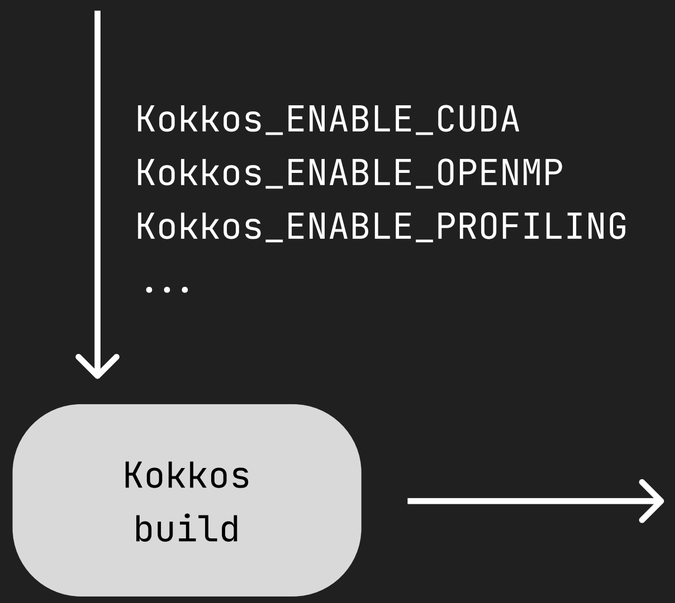
CPU/GPU cluster
(e.g., nvc + icc compilers)

Philosophy behind Kokkos

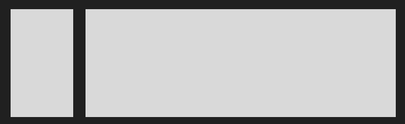
```
#include <complex.h>
#include <stdio.h>
#include <time.h>
int *d, D[9999], N=20, L=4, n, m, k, a[3], i;char *p, *q, S[2000]="L@X-SGD-HNBBB-AD-VHSG--XNT\x1b[2J\x1b[H",
*s=S, *G="r2zZX!.+@KBK^yh.!:%Bud!.+Jyh.!6.BHBhp!6.BHBh!:%Buv{VT!.hBJ6p!042ljn!284b}`!.hR6Dp!.hp!h.T6p!h.p6!
2/LilqP72!h.+@QB!~}lqP72/Lil!h.+@QBFp!:)0?F]nwf!,82v!.sv{6!.l6!,j<n8!.xN6t!&NvN*!.6hp";
/*nqncgrqsebzlhfhxrraqbujjvggre:@znzrggre*/typedef complex double(c);c(X)[3], P, O;c B(double t) {double s=1-
t, u;P=s*s*X[1]+2*s*t**X+t*t*X[2]+O;u=I*P;return +48*((s=P)+48*I)/(1<u?u:1);}void b(double t, double u)
{double s=P=B(t)-B(u);(s=P*(2*s-P))<1?m=P=B((t+u)/2), k=-I*P,m > -41 &&m<39 &&9< k &&k<48?
m+=k/2*80+73,S[m]=S[m]-73?k %2? S[m]-94?95:73:S[m]-95?94:73:73: 1: (b(t, (t+u)/2), b((t+u)/2, u), 0);}int
main(int(x), char **V) {int aug=0, augn=59;const char
gammds[]="ARIRE\nTBAAN\nTVIR\nLBH\nHC";clock_t(c)=clock();for (d=D; m<26; m++, s++)*s > 63?*d++=m %7*
16-7*8, *d++=m/7*25, *d++=*s-64:0;for (d=D, L=N=m=0; aug<augn; aug++) {x=gammds[aug];if (x > 13?64<x &&x<91?
*d++=m*16, *d++=L*25,*d++=x % 26:0, m++,0: 1)for (++L; d-D > N*3||(m=0); N++)D[N*3] -= m*8;}for (;
i<200+L*25; i++) {for (n=0, p=S+33; n<1920; *p++=n++ % 80> 78?10:32) {}for (*p=x=0, d=D; x<N; x++, d+=3)
{O=(d[1]-i-40)*I+*d;n=d[2];p=G;for (; n--;) {for (; *p++ > 33;) {}}*a=a[1]=*p++;for (; *p > 33; p++)if (*p
%2? *a=*p,0: 1) {a[2]=*p;for (m=0; m<3; m++) {k=a[m]/2-18;q="/&&&##%%##.+),A$$$$'&&'&&((%-((#'/#%
%#&#\&&#D&";for (n=2; k--;)n+=*q++-34;X[m]=n % 13+n/13*I;}b(0, 1);*a=a[1]=*p;}}for (puts(s), s=S+30;
(clock()-c)*20<i*CLOCKS_PER_SEC;) {}}return 0;}
```

as scientists we want to focus
on writing
science code
not housekeeping or porting

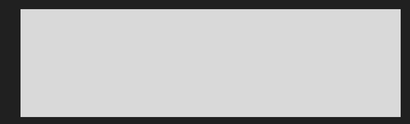
kokkos::View
kokkos::RangePolicy
kokkos::parallel_for
...



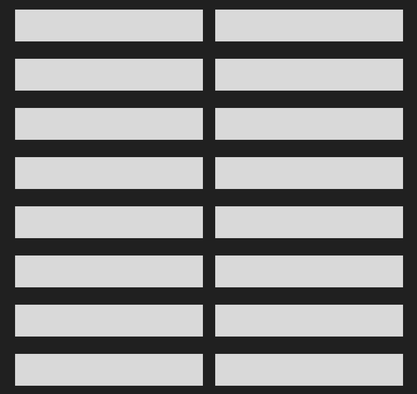
PC with a GPU
(e.g., nvcc + gcc)



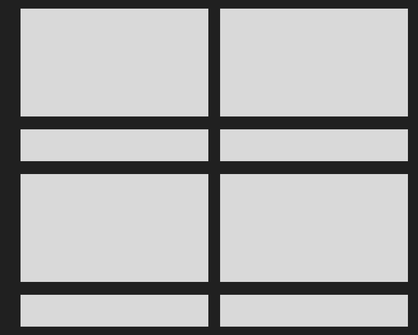
Personal laptop
(e.g., llvm compiler)



CPU cluster
(e.g., AOCC compiler)



CPU/GPU cluster
(e.g., nvc + icc compilers)



CUDA code

```cpp
// define sizes
auto   N        = (std::size_t)(1e8);
auto   mem_size = (std::size_t)(N * sizeof(float));

// initialize/allocate device memory
float* A = nullptr;
float* B = nullptr;
float* C = nullptr;
cudaMalloc((void**)&A, mem_size);
cudaMalloc((void**)&B, mem_size);
cudaMalloc((void**)&C, mem_size);

// allocate host memory
auto A_h = new float[N];
auto B_h = new float[N];
auto C_h = new float[N];

// initialize host memory with random numbers [0, 1[
for (std::size_t i = 0; i < N; ++i) {
  A_h[i] = rand() / (float)RAND_MAX;
  B_h[i] = rand() / (float)RAND_MAX;
}

// copy host memory to device
cudaMemcpy(A, A_h, mem_size, cudaMemcpyHostToDevice);
cudaMemcpy(B, B_h, mem_size, cudaMemcpyHostToDevice);

// ...
```

```cpp
// ...

__global__ void vectorAdd(const float* A,
                          const float* B,
                          float* C,
                          int N) {
  int i = blockDim.x * blockIdx.x + threadIdx.x;
  if (i < N) {
    C[i] = A[i] + B[i];
  }
}

// ...

int threadsPerBlock = 512;
int blocksPerGrid   = (N + threadsPerBlock - 1) / threadsPerBlock;
vectorAdd<<<blocksPerGrid, threadsPerBlock>>>(A, B, C, N);
cudaDeviceSynchronize();
```

CUDA code

```cpp
// define sizes
auto   N        = (std::size_t)(1e8);
auto   mem_size = (std::size_t)(N * sizeof(float));

// initialize/allocate device memory
float* A = nullptr;
float* B = nullptr;
float* C = nullptr;
cudaMalloc((void**)&A, mem_size);
cudaMalloc((void**)&B, mem_size);
cudaMalloc((void**)&C, mem_size);

// allocate host memory
auto A_h = new float[N];
auto B_h = new float[N];
auto C_h = new float[N];

// initialize host memory with random numbers [0, 1[
for (std::size_t i = 0; i < N; ++i) {
  A_h[i] = rand() / (float)RAND_MAX;
  B_h[i] = rand() / (float)RAND_MAX;
}

// copy host memory to device
cudaMemcpy(A, A_h, mem_size, cudaMemcpyHostToDevice);
cudaMemcpy(B, B_h, mem_size, cudaMemcpyHostToDevice);
```

Kokkos code

```cpp
// alias for the array type
using array_t = Kokkos::View<float*>;
auto N        = (std::size_t)(1e8);

// declare/allocate the views (arrays)
array_t A("A", N);
array_t B("B", N);
array_t C("C", N);

// create host mirrors of the views
// (when compiled on CPU the memory is *not* duplicated)
auto A_h = Kokkos::create_mirror_view(A);
auto B_h = Kokkos::create_mirror_view(B);
auto C_h = Kokkos::create_mirror_view(C);

// initialize arrays on the host
for (std::size_t i {0}; i < N; ++i) {
  A_h(i) = rand() / (float)RAND_MAX;
  B_h(i) = rand() / (float)RAND_MAX;
}

// copy the host arrays to the device
// (when compiled on CPU this is a no-op)
Kokkos::deep_copy(A, A_h);
Kokkos::deep_copy(B, B_h);
```

CUDA code

```cpp
// define sizes
auto   N        = (std::size_t)(1e8);
auto   mem_size = (std::size_t)(N * sizeof(float));

// initialize/allocate device memory
float* A = nullptr;
float* B = nullptr;
float* C = nullptr;
cudaMalloc((void**)&A, mem_size);
cudaMalloc((void**)&B, mem_size);
cudaMalloc((void**)&C, mem_size);

// allocate host memory
auto A_h = new float[N];
auto B_h = new float[N];
auto C_h = new float[N];

// initialize host memory with random numbers [0, 1[
for (std::size_t i = 0; i < N; ++i) {
  A_h[i] = rand() / (float)RAND_MAX;
  B_h[i] = rand() / (float)RAND_MAX;
}

// copy host memory to device
cudaMemcpy(A, A_h, mem_size, cudaMemcpyHostToDevice);
cudaMemcpy(B, B_h, mem_size, cudaMemcpyHostToDevice);
```

Kokkos code

```cpp
// alias for the array type
using array_t = Kokkos::View<float*>;
auto N        = (std::size_t)(1e8);

// declare/allocate the views (arrays)
array_t A("A", N);
array_t B("B", N);
array_t C("C", N);

// create host mirrors of the views
// (when compiled on CPU the memory is *not* duplicated)
auto A_h = Kokkos::create_mirror_view(A);
auto B_h = Kokkos::create_mirror_view(B);
auto C_h = Kokkos::create_mirror_view(C);

// initialize arrays on the host
for (std::size_t i {0}; i < N; ++i) {
  A_h(i) = rand() / (float)RAND_MAX;
  B_h(i) = rand() / (float)RAND_MAX;
}

// copy the host arrays to the device
// (when compiled on CPU this is a no-op)
Kokkos::deep_copy(A, A_h);
Kokkos::deep_copy(B, B_h);
```

CUDA code

```cpp
__global__ void vectorAdd(const float* A,
                          const float* B,
                          float* C,
                          int N) {
  int i = blockDim.x * blockIdx.x + threadIdx.x;
  if (i < N) {
    C[i] = A[i] + B[i];
  }
}

// ...

int threadsPerBlock = 512;
int blocksPerGrid   = (N + threadsPerBlock - 1) / threadsPerBlock;
vectorAdd<<<blocksPerGrid, threadsPerBlock>>>(A, B, C, N);
cudaDeviceSynchronize();
```

Kokkos code

```cpp
struct VectorAdd {
  VectorAdd(const array_t& A,
            const array_t& B,
            const array_t& c)
  : a(A), b(B), c(C) {}
  KOKKOS_INLINE_FUNCTION
  void operator()(index_t i) const {
    c(i) = a(i) + b(i);
  }
private:
  array_t a, b, c;
};

// ...

Kokkos::parallel_for("VectorAdd", N, VectorAdd(a, b, c));
```

CPU/GPU portability of Kokkos

- Memory layout

```
// 1d array of doubles of length N
// ... allocated on the host (CPU)
Kokkos::View<double*, Kokkos::HostSpace> A1("A1", N);

// 2d array of doubles of N x 3(compile time) dimension
// ... allocated on the device (GPU)
Kokkos::View<double*[3], Kokkos::CudaSpace> A2("A2", N);

// 3d array of ints of N x M x 6(compile time) dimension
// ... allocated on either the device or the host
Kokkos::View<int**[6], Kokkos::DefaultExecutionSpace::memory_space> A3("A3", N, M);

// 1d array of ints of length N
// ... allocated on the default exec space
Kokkos::View<int*, Kokkos::LayoutLeft> A4("A4", N);

// mirror of A4 on the host (pointer copy when both on host)
// ... automatically picks the correct layout/dimensions
auto A4_h = Kokkos::create_mirror_view(A4);
```

CPU/GPU portability of Kokkos

- Execution space (policy)

```
struct SomeFunctor {
  KOKKOS_INLINE_FUNCTION
  void operator()(index_t i) const {
    C(i) = A(i) + B(i);
  }
private:
  // ...
};
```

```
// Compiles into a 1D for loop optimized with OpenMP
// ... automatically picks the optimal data traversal layout
// ... assumes A, B, C live on the host
Kokkos::parallel_for("my openmp loop",
                     Kokkos::RangePolicy<Kokkos::OpenMP>(0, N),
                     SomeFunctor(A, B, C));

// Compiles into a 2D for loop on the GPU
// ... assumes A, B, C already exist on the device
Kokkos::parallel_for("my CUDA loop",
                     Kokkos::MDRangePolicy<Kokkos::Rank<2>,
                                           Kokkos::Cuda>({0, 0}, {Nx, Ny}),
                     SomeFunctor(A, B, C));
```

Building with Kokkos

- Building with CMake

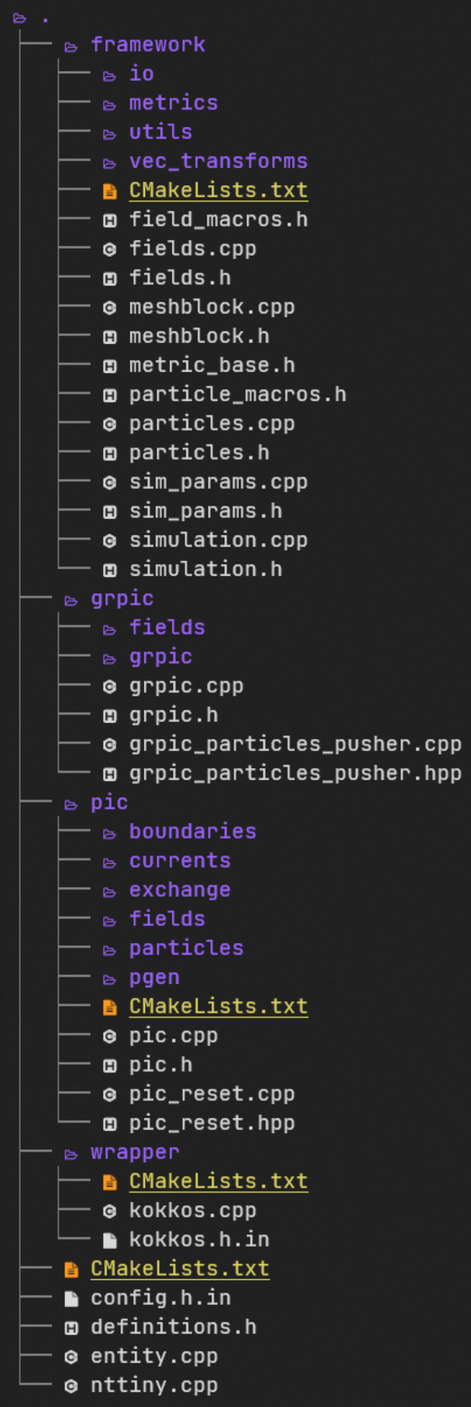
```
add_subdirectory(path_to/kokkos kokkos)
target_link_libraries(${YOUR_TARGET} PRIVATE Kokkos::kokkos)
```

- Building with Makefile

```
// define Kokkos flags
include $(KOKKOS_PATH)/Makefile.kokkos

// Kokkos C++ flags
CXX_FLAGS += $(KOKKOS_CPPFLAGS) $(KOKKOS_CXXFLAGS)
// add Kokkos objects to dependencies
OBJS += $(KOKKOS_LINK_DEPENDS)
// add Kokkos ldflags
LDFLAGS += $(KOKKOS_LDFLAGS)
// add Kokkos libs
LIBS += $(KOKKOS_LIBS)
// add Kokkos sources
SRCS += $(KOKKOS_CPP_DEPENDS)
```

Building with Kokkos



Building with Kokkos

most of the code
knows nothing about
Kokkos!

```cpp
struct Fields {
  ndfield_t<D, 6>         em;
  ndfield_mirror_t<D, 6> em_h;
}

// ...

template <Dimension D>
auto rangeActiveCells() -> range_t<D>;
```

Building with Kokkos

```
.
├── framework
│   ├── io
│   ├── metrics
│   ├── utils
│   ├── vec_transforms
│   ├── CMakeLists.txt
│   ├── field_macros.h
│   ├── fields.cpp
│   ├── fields.h
│   ├── meshblock.cpp
│   ├── meshblock.h
│   ├── metric_base.h
│   ├── particle_macros.h
│   ├── particles.cpp
│   ├── particles.h
│   ├── sim_params.cpp
│   ├── sim_params.h
│   ├── simulation.cpp
│   └── simulation.h
├── grpic
│   ├── fields
│   ├── grpic
│   ├── grpic.cpp
│   ├── grpic.h
│   ├── grpic_particles_pusher.cpp
│   └── grpic_particles_pusher.hpp
├── pic
│   ├── boundaries
│   ├── currents
│   ├── exchange
│   ├── fields
│   ├── particles
│   ├── pgen
│   ├── CMakeLists.txt
│   ├── pic.cpp
│   ├── pic.h
│   ├── pic_reset.cpp
│   └── pic_reset.hpp
├── wrapper
│   ├── CMakeLists.txt
│   ├── kokkos.cpp
│   └── kokkos.h.in
├── CMakeLists.txt
├── config.h.in
├── definitions.h
├── entity.cpp
└── nttiny.cpp
```

```cpp
template <typename T>
using array_t = Kokkos::View<T, AccelMemSpace>;

template <Dimension D, int N>
using ndfield_t = typename std::conditional<
  D == Dim1,
  array_t<real_t* [N]>,
  typename std::conditional<
    D == Dim2,
    array_t<real_t** [N]>,
    typename std::conditional<D == Dim3, array_t<real_t*** [N]>, std::nullptr_t>::type>::
    type>::type;

template <Dimension D>
using range_t = typename std::conditional<
  D == Dim1,
  Kokkos::RangePolicy<AccelExeSpace>,
  typename std::conditional<
    D == Dim2,
    Kokkos::MDRangePolicy<Kokkos::Rank<2>, AccelExeSpace>,
    typename std::conditional<D == Dim3,
                              Kokkos::MDRangePolicy<Kokkos::Rank<3>, AccelExeSpace>,
                              std::nullptr_t>::type>::type>::type;
```

wrapper that
defines aliases

Final thoughts

1. Avoid writing pure CUDA!

    (a) you are very likely to do it wrong
    (b) debugging is a nightmare
    (c) your code is stuck to only run on NVIDIA GPUs
    (d) you'll have to reconfigure the parameters depending on the microarchitecture of a particular GPU for most performance

Final thoughts

1. Avoid writing pure CUDA!
2. Kokkos is almost as fast as it gets

the performance (in most scenarios) is almost as fast as a well-optimized CUDA/OpenACC code

```
C[i] = A[i] + B[i] on 1e8 elements:
Serial (CPU) time: 135.246 ms
OpenMP (CPU) time:  21.629 ms
OpenACC (GPU) time:  1.488 ms
CUDA time:           1.256 ms
Kokkos (GPU) time:   1.684 ms
```

```
Bins[Particles[i]] += 1 on 1e8 particles
and 1e6 bins:
Serial (CPU) time: 346.089 ms
OpenMP (CPU) time: 421.898 ms
OpenACC (GPU) time:  2.309 ms
CUDA time:           2.488 ms
Kokkos (GPU) time:   2.525 ms
```

source code available: https://github.com/haykh/princeton-bootcamp-kokkos-2022

Final thoughts

1. Avoid writing pure CUDA!
2. Kokkos is almost as fast as it gets
3. Is it worth it?
   a. are you ok to rely on proprietary standard which might or might not go away in a few years?
   b. is your code well-established?
   c. are you interested in the libraries of Kokkos ecosystem?
   d. how wide of a user-base do you expect (e.g., vastly different architectures)?
   e. with Kokkos it’s easier to write wrappers and abstract against the architecture

Resources

1. GitHub: github.com/kokkos/kokkos
2. Wiki: kokkos.github.io/kokkos-core-wiki
3. Video lectures: youtube.com/watch?v=rUIcWtFU5qM
4. Kokkos tutorials: github.com/kokkos/kokkos-tutorials
5. Kokkos team slack: kokkosteam.slack.com
6. + Kokkos kernels + Kokkos tools: kokkos.github.io