Performance portability for integral kernels in GAMESS

2020 Performance, Portability, and Productivity in HPC Forum

Sept. 1, 2020

Colleen Bertoni,

Performance Engineering Group, ALCF

Acknowledgements

- GAMESS team
  - Giuseppe Barca,
  - Jorge Galvez Vallejo
  - David Poole
- ECP OpenMP hackathon
  - Michael Kruse
  - Ye Luo

This research was supported by the Exascale Computing Project (17-SC-20-SC), a joint project of the U.S. Department of Energy's Office of Science and National Nuclear Security Administration, responsible for delivering a capable exascale ecosystem, including software, applications, and hardware technology, to support the nation's exascale computing imperative.

We gratefully acknowledge the computing resources provided and operated by the Joint Laboratory for System Evaluation (JLSE) at Argonne National Laboratory.

Motivation and Objective

Objective

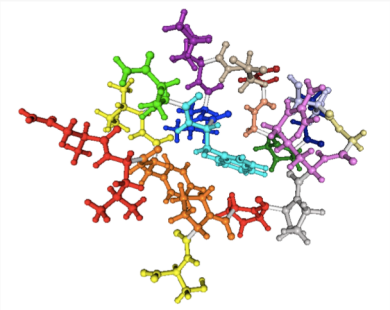
- CUDA is a proprietary language that is not device agnostic or portable
  - Not supported on Aurora or Frontier
- For the CUDA code in the standalone Hartree-Fock application:
  1. How difficult it was to switch from CUDA to other programming models?
  2. After changes, how does the performance compare?

GAMESS Intro

Background and overview of code

- GAMESS is the General Atomic and Molecular Electronic Structure System
  - General-purpose electronic structure code (many methods and capabilities)
  - ~1 million lines of Fortran
  - Began in 1980s in Mark Gordon's group at North Dakota State, and originally forking off of HONDO 5, an NSF and DOE funded project
- Optional C/C++ GPU-accelerated libraries/applications in GAMESS
  - Standalone version which computes only Hartree-Fock energy for fragmented systems

Current Parallelization

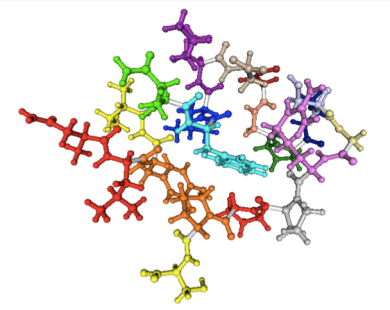


$E_{HF} = 0;$
forall *fragments and fragment pairs* **do**
  Guess initial $\boldsymbol{D}$ matrix;
  Compute $\boldsymbol{H}^{core}$;
  $\boldsymbol{F} = \boldsymbol{H}^{core}$;
  repeat
    for *ERI batches* $\{ab|cd\}$ **do**
      Compute ERI $(\alpha\beta|\gamma\delta) \in \{ab|cd\}$;
      $F_{\alpha\beta} \mathrel{+}= \sum_{\gamma\delta} D_{\gamma\delta} \left[(\alpha\beta|\gamma\delta) - \frac{1}{2}(\alpha\delta|\gamma\beta)\right]$;
    end
    Diagonalize $\boldsymbol{F}$ and obtain new $\boldsymbol{D}$;
  until *converged*;
  $E_{HF} \mathrel{+}= \frac{1}{2}\sum_{\alpha\beta}(H^{core}_{\alpha\beta} + F_{\alpha\beta})$;
end
Subtract fragment energies from E_{HF};

- Code is C/C++, MPI+CUDA, targeting multiply GPUs per node
- Levels of concurrency:
  - Fragments and fragment pairs in the molecular systems
  - Electron repulsion integral (ERI) computation and digestion

Barca, et. al, “Scaling the Hartree-Fock method on Summit” SC ‘20

Current Parallelization

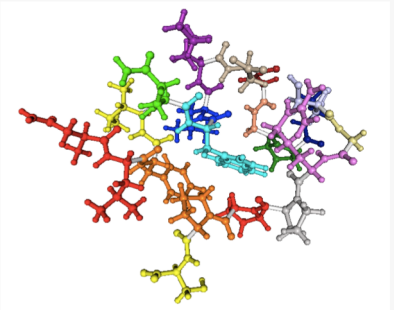


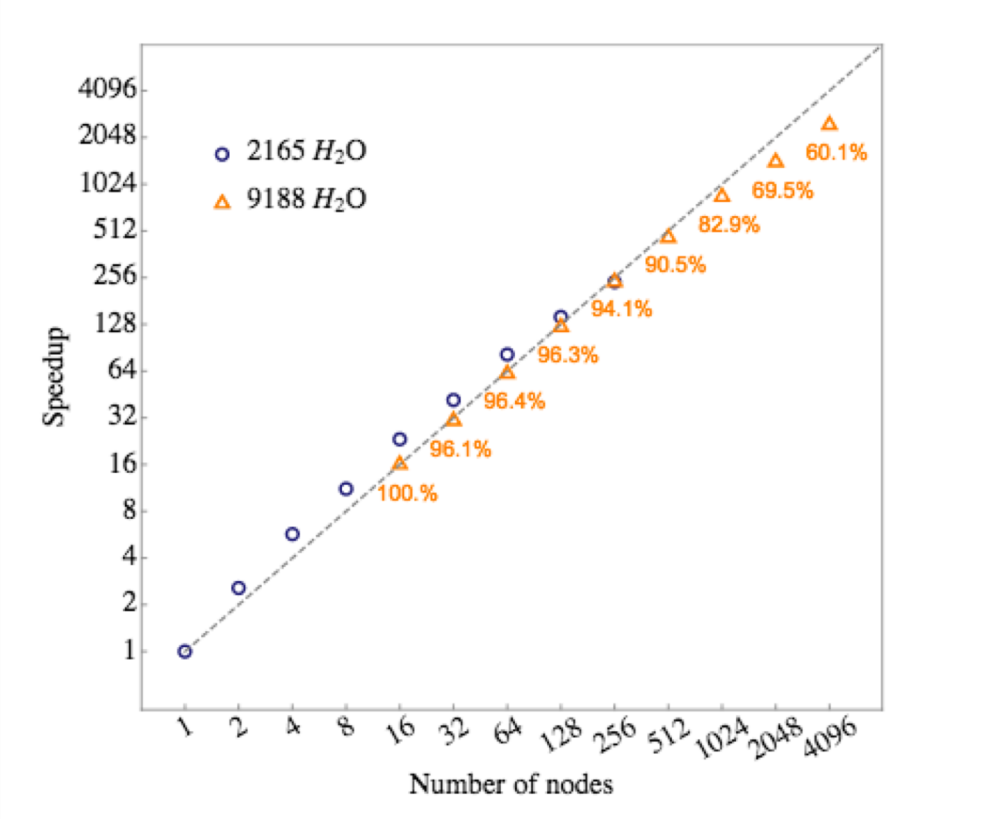
$E_{HF} = 0$;
forall *fragments and fragment pairs* **do**
  Guess initial $\boldsymbol{D}$ matrix;
  Compute $\boldsymbol{H}^{core}$;
  $\boldsymbol{F} = \boldsymbol{H}^{core}$;
  repeat
    for *ERI batches* $\{ab|cd\}$ **do**
      Compute ERI $(\alpha\beta|\gamma\delta) \in \{ab|cd\}$;
      $F_{\alpha\beta} \mathrel{+}= \sum_{\gamma\delta} D_{\gamma\delta} \left[(\alpha\beta|\gamma\delta) - \frac{1}{2}(\alpha\delta|\gamma\beta)\right]$;
    end
    Diagonalize $\boldsymbol{F}$ and obtain new $\boldsymbol{D}$;
  until *converged*;
  $E_{HF} \mathrel{+}= \frac{1}{2}\sum_{\alpha\beta}(H^{core}_{\alpha\beta} + F_{\alpha\beta})$;
end
Subtract fragment energies from E_{HF};

- Code is C/C++, MPI+CUDA, targeting multiply GPUs per node
- Levels of concurrency:
  - Fragments and fragment pairs in the molecular systems (MPI)
  - Electron repulsion integral (ERI) computation and digestion (MPI+CUDA)

Barca, et. al, “Scaling the Hartree-Fock method on Summit” SC ‘20

Current Parallelization





- Code is C/C++, MPI+CUDA, targeting multiply GPUs per node
- Levels of concurrency:
  - Fragments and fragment pairs in the molecular systems (MPI)
  - Electron repulsion integral (ERI) computation and digestion (MPI+CUDA)

Barca, et. al, “Scaling the Hartree-Fock method on Summit” SC ‘20

Porting Strategy

Porting Strategy

CUDA calls (# times in source)
cudaMemcpy (9) cuda{Malloc,Free} (50) cuda{Malloc/Free}Host (12) cudaMemset (4) cudaMemcpyAsync (15) cudaMemsetAsync (60)
cublasDgemm (5) cublasDdot (1) cublasDscal (1)
hand_written_kernel<<<…>>>(…) (529) (77 total kernels)

Porting Strategy

CUDA calls (# times in source)	
cudaMemcpy (9) cuda{Malloc,Free} (50) cuda{Malloc/Free}Host (12) cudaMemset (4) cudaMemcpyAsync (15) cudaMemsetAsync (60)	Convert MemoryPool class to alternative programming models
cublasDgemm (5) cublasDdot (1) cublasDscal (1)	Leave as is, we want to use vendor math libraries
hand_written_kernel<<<…>>>(…) (529) (77 total kernels)	Rewrite in alternative programming models. CUDA kernels are generated by a script, so this will involve adjusting the code generator

Porting Strategy

CUDA calls (# times in source)
cudaMemcpy (9) cuda{Malloc,Free} (50) cuda{Malloc/Free}Host (12) cudaMemset (4) cudaMemcpyAsync (15) cudaMemsetAsync (60)
cublasDgemm (5) cublasDdot (1) cublasDscal (1)
hand_written_kernel<<<…>>>(…) (529) (77 total kernels)

- Extract a hand written integral CUDA kernel
  - (ss|ss)
- Port to other programming models:
  - HIP
  - OpenMP
  - DPC++

Evaluation of ERIs

$$(ij|kl) = \int\int \phi_i(r_1)\phi_j(r_1)\frac{1}{r_{12}}\phi_k(r_2)\phi_l(r_2)dr_1dr_2$$

```
// computes the ERI(i,j,k,l)
for(int ab = 0; a < K_i; i++){
   for(int b = 0; b < K_j; j++){
     for(int c = 0; c < K_k; k++){
       for(int d = 0; d < K_l; l++){
         double coefficient = Kab_gl(...)*Kcd_gl(...);
         ERI_array[i][j][k][l] += coefficient * boys_function(..)
       }
     }
   }
}
```

- Each integral is written in terms of the generalized Boys function
- Computation of Boys function is done via modified cubic Chebyshev interpolation

Evaluation of ERIs

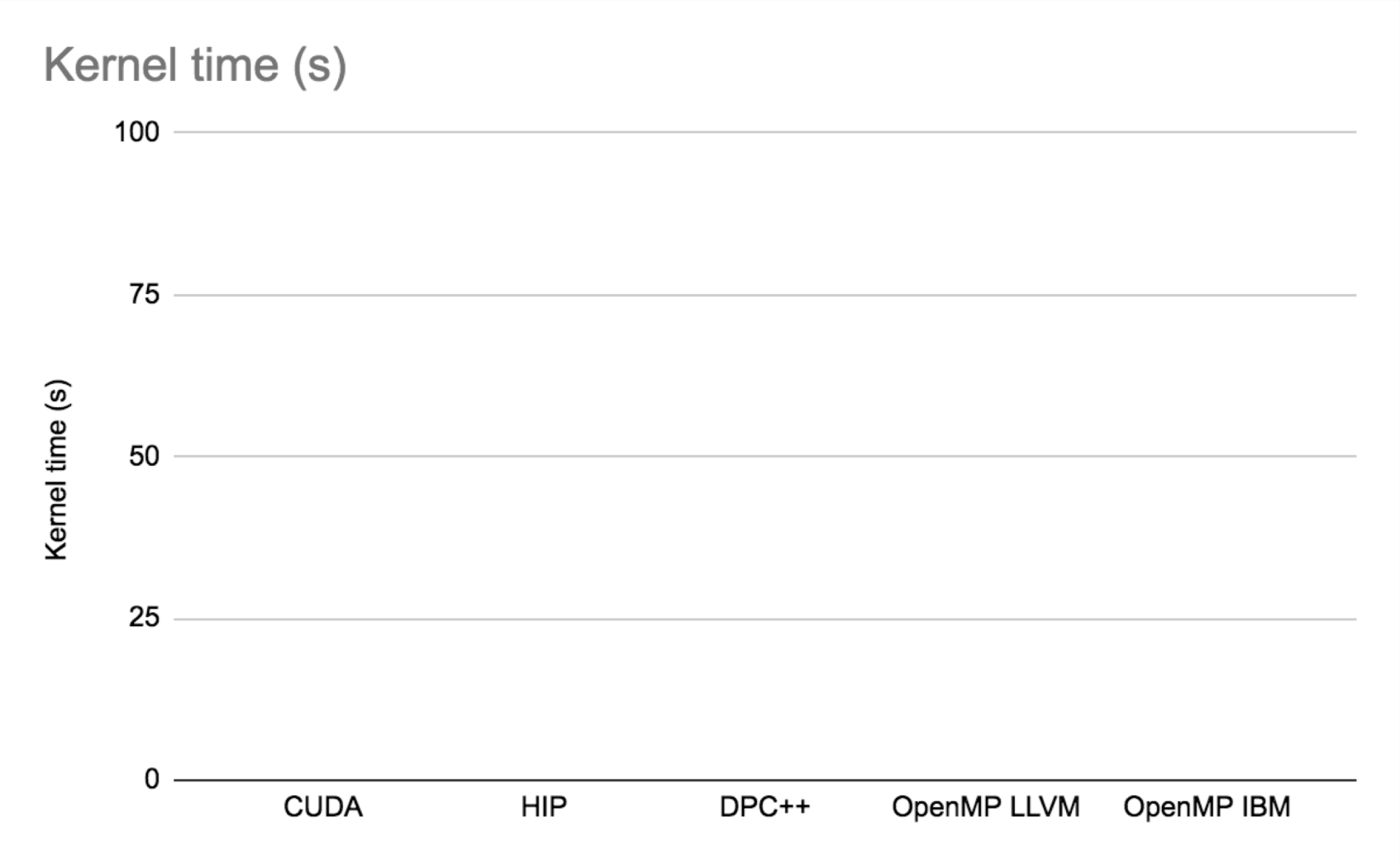
$$(00|00) = \sum_{a}^{K_i}\sum_{b}^{K_j}\sum_{c}^{K_k}\sum_{d}^{K_l} U_{abcd,ijkl} F_0(T_{abcd})$$

```
// computes the ERI(i,j,k,l)
for(int ab = 0; a < K_i; i++){
   for(int b = 0; b < K_j; j++){
     for(int c = 0; c < K_k; k++){
       for(int d = 0; d < K_l; l++){
         double coefficient = Kab_gl(...)*Kcd_gl(...);
         ERI_array[i][j][k][l] += coefficient * boys_function(..)
       }
     }
   }
}
```

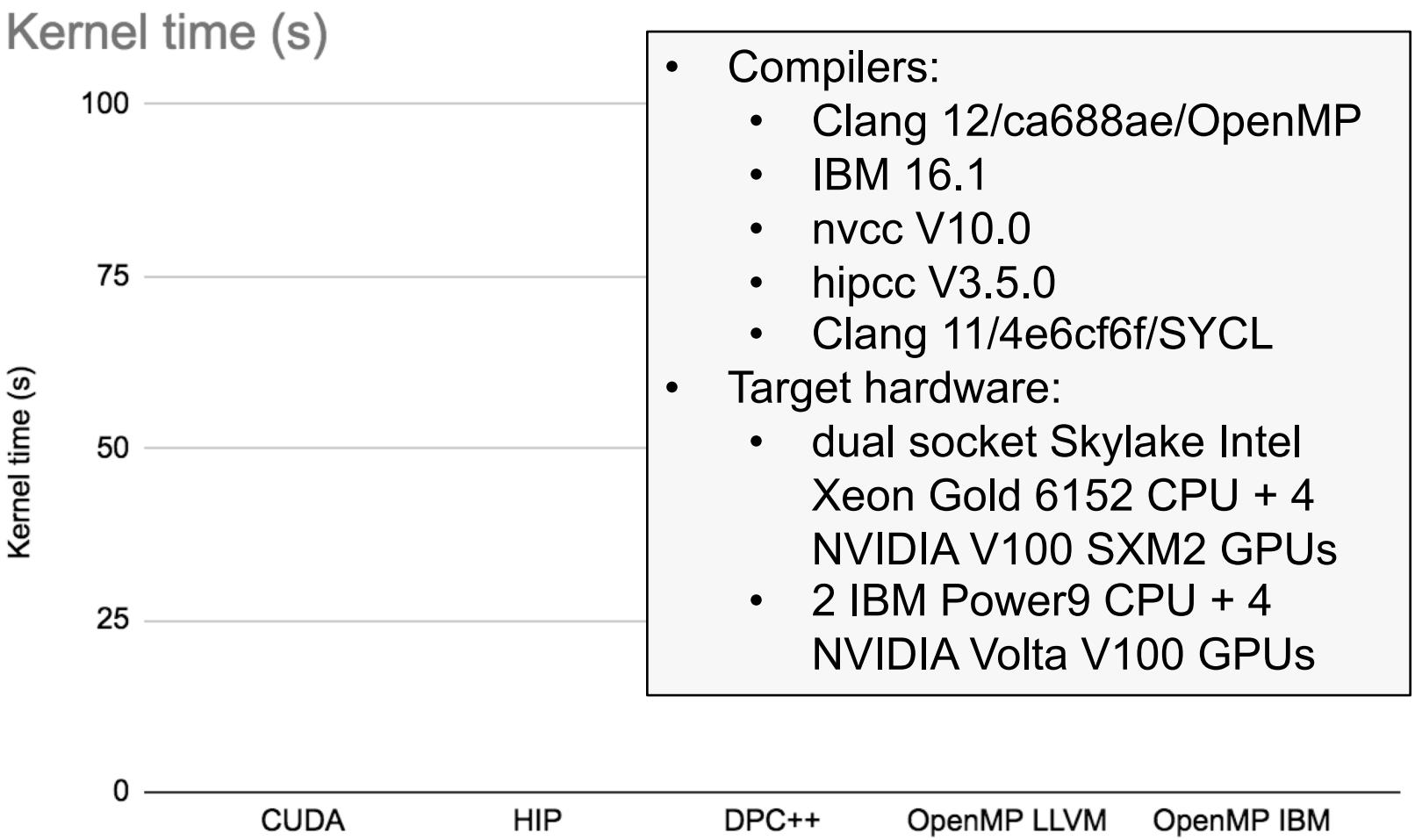
- Each integral is written in terms of the generalized Boys function
- Computation of Boys function is done via modified cubic Chebyshev interpolation

Code Comparison

Code comparison: timing



Code comparison: timing



- Compilers:
  - Clang 12/ca688ae/OpenMP
  - IBM 16.1
  - nvcc V10.0
  - hipcc V3.5.0
  - Clang 11/4e6cf6f/SYCL
- Target hardware:
  - dual socket Skylake Intel Xeon Gold 6152 CPU + 4 NVIDIA V100 SXM2 GPUs
  - 2 IBM Power9 CPU + 4 NVIDIA Volta V100 GPUs

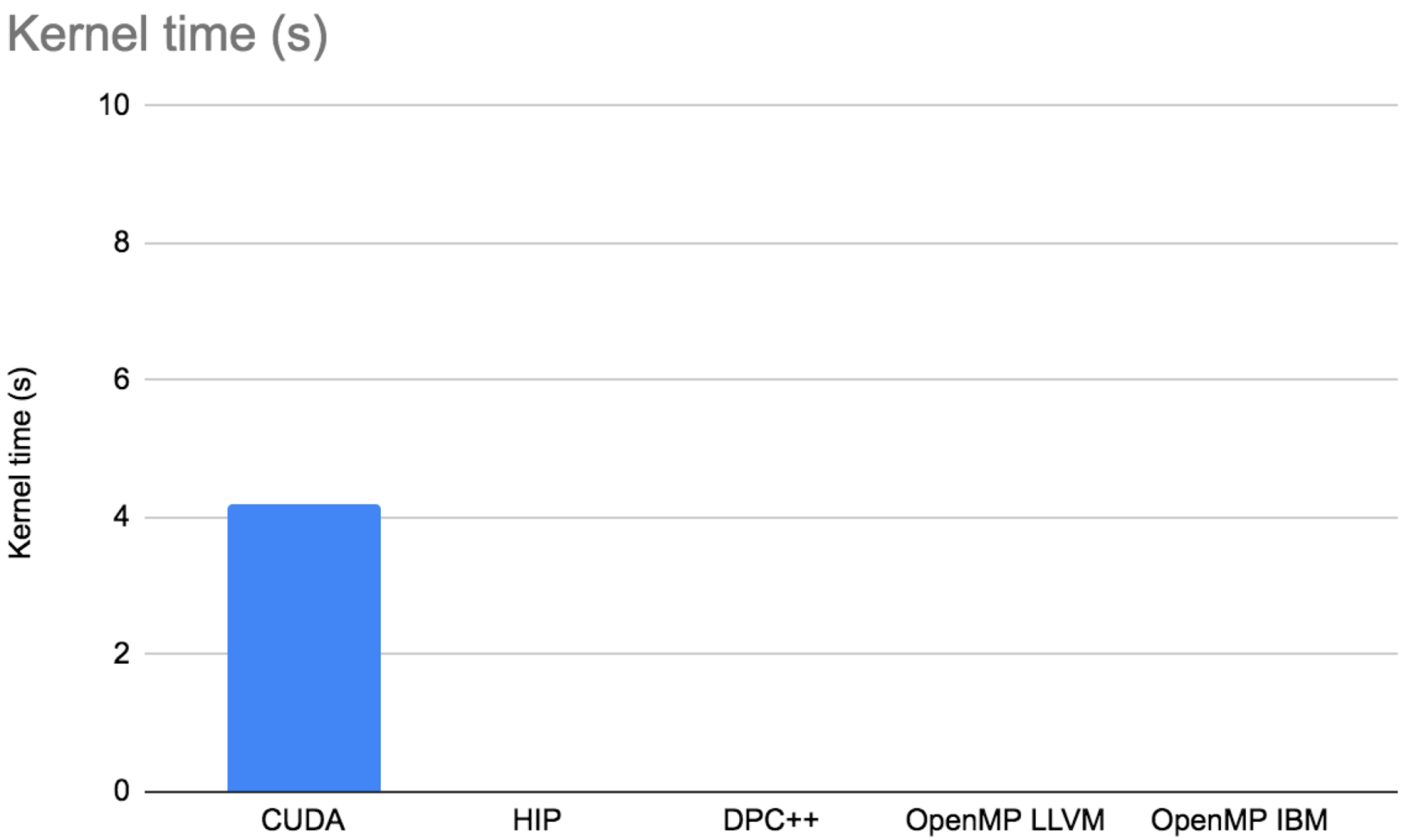
```
cudaMemcpy( d_Kab_array, h_Kab_array, size, cudaMemcpyDeviceToHost );
...
kernel_0000<<<grid_d,block_d>>>( d_Kab_array, ..., ERI_array);
```

launch appropriate number of thread blocks to compute with num_ab_shells*num_cd_shells with block size 128

```
__global__ void kernel_0000( int* __restrict__ Kab_gl,
                             double*  __restrict__ ERI_array ){
  int ab_index = blockIdx.x/n_blocks_cd;
  int cd_index = threadIdx.x + (cd_block) * BlockDim.x;

  __shared__ double UP_sh[MAX_KAB], ... ;

  UP_sh[threadIdx.x] = UP_gl[ab_index + threadIdx.x*nab];
  ...

  for(unsigned int kcd = 0; kcd < contraction_cd; ++kcd){
    const double UQ = UQ_gl[cd_index+ncd*kcd];
    ...
      for(unsigned int kab = 0; kab < contraction_ab; ++kab){
        const double UP = UP_sh[kab];
        ...
          m[0] = boysf(..);
          ERI_array[ab_index*(stride)+ cd_index] += m[0]*theta;
      }
  }
}
```

Computation is partitioned so each thread executes the kernel and accumulates into ERI_array, indexed by the thread

Input: (ss|ss) , 200 water cluster, PC-seg-0



Hardware: dual socket Skylake Intel Xeon Gold 6152 CPU + 4 NVIDIA V100 SXM2 GPUs
IBM Power9 (2 Power9 CPU + 4 NVIDIA Volta V100 GPUs)

CUDA to HIP

```
cudaMemcpy( d_Kab_array, h_Kab_array, size, cudaMemcpyDeviceToHost );
...
kernel_0000<<<grid_d,block_d>>>( d_Kab_array, ..., ERI_array);
```

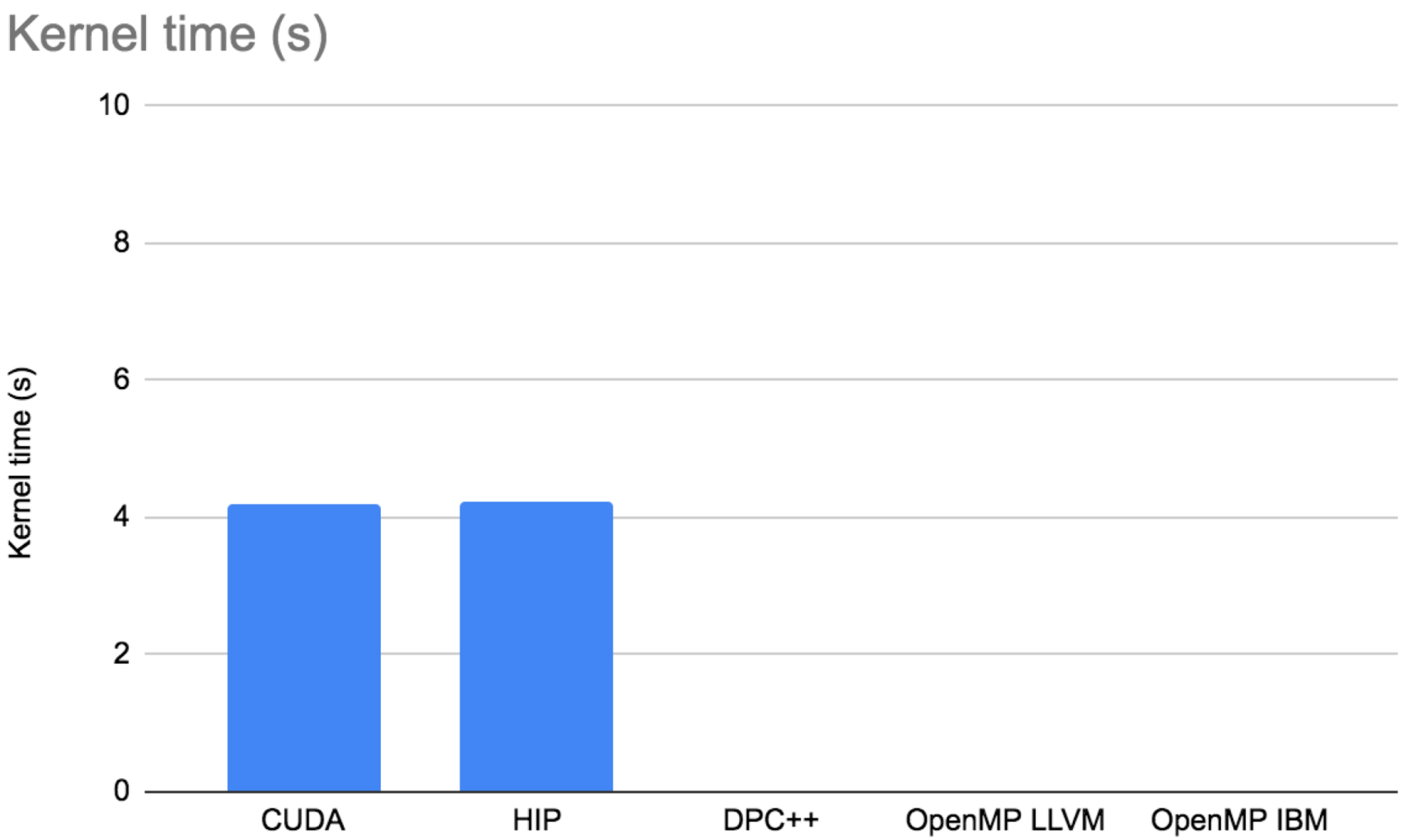
```
__global__ void kernel_0000( int* __restrict__ Kab_gl, ... ,
                             double*  __restrict__ ERI_array ){
  int ab_index = blockIdx.x/n_blocks_cd;
  int cd_index = threadIdx.x + (cd_block) * BlockDim.x;

  __shared__ double UP_sh[MAX_KAB], ... ;

  UP_sh[threadIdx.x] = UP_gl[ab_index + threadIdx.x*nab];
  ...

  for(unsigned int kcd = 0; kcd < contraction_cd; ++kcd){
    const double UQ = UQ_gl[cd_index+ncd*kcd];
    ...
      for(unsigned int kab = 0; kab < contraction_ab; ++kab){
        const double UP = UP_sh[kab];
        ...
          m[0] = boysf(..);
          ERI_array[ab_index*(stride)+ cd_index] += m[0]*theta;
      }
  }
}
```

```
hipMemcpy( d_Kab_array, h_Kab_array, size, hipMemcpyHostToDevice );
...
hipLaunchKernelGGL(kernel_0000, dim3(grid_d),
                   dim3(block_d) d_Kab_array, ...,
                   ERI_array);
```

```
__global__ void kernel_0000( int* __restrict__ Kab_gl, ... ,
                             double*  __restrict__ ERI_array ){
  int ab_index = hipBlockIdx_x/n_blocks_cd;
  int cd_index = hipThreadId_x + (cd_block) * hipBlockDim_x;

  __shared__ double UP_sh[MAX_KAB], ... ;

  UP_sh[hipBlockIdx_x] = UP_gl[ab_index + hipBlockIdx_x *nab];
  ...

  for(unsigned int kcd = 0; kcd < contraction_cd; ++kcd){
    const double UQ = UQ_gl[cd_index+ncd*kcd];
    ...
      for(unsigned int kab = 0; kab < contraction_ab; ++kab){
        const double UP = UP_sh[kab];
        ...
          m[0] = boysf(..);
          ERI_array[ab_index*(stride)+ cd_index] += m[0]*theta;
      }
  }
}
```

Porting was very simple from CUDA → HIP for the kernel

Input: (ss|ss) , 200 water cluster, PC-seg-0



Hardware: dual socket Skylake Intel Xeon Gold 6152 CPU + 4 NVIDIA V100 SXM2 GPUs
IBM Power9 (2 Power9 CPU + 4 NVIDIA Volta V100 GPUs)

CUDA to OpenMP

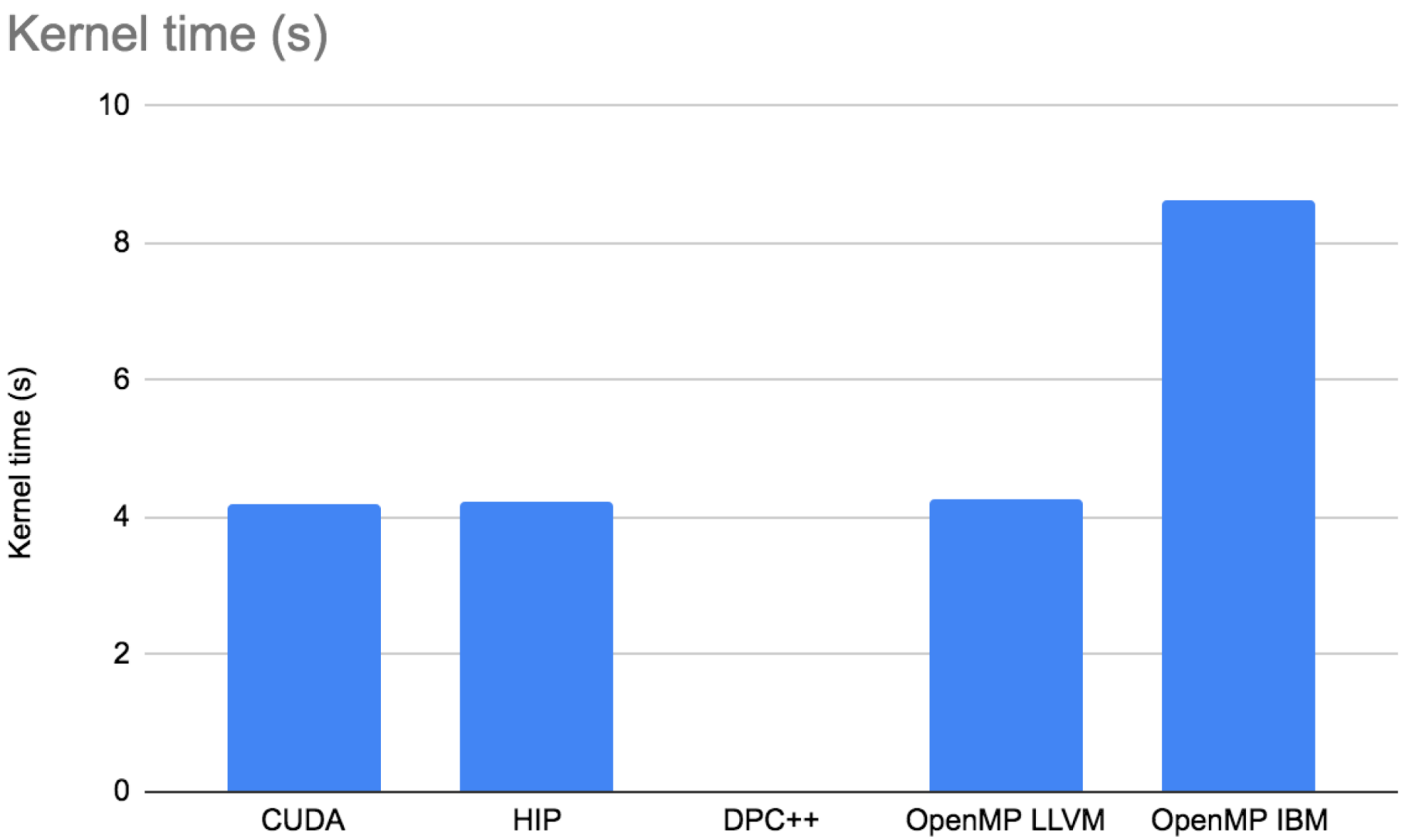
```
cudaMemcpy( d_Kab_array, h_Kab_array, size, cudaMemcpyDeviceToHost );
...
kernel_0000<<<grid_d,block_d>>>( d_Kab_array, ..., ERI_array);
```

```
__global__ void kernel_0000( int* __restrict__ Kab_gl, ... ,
                             double*  __restrict__ ERI_array ){


  int ab_index = blockIdx.x/n_blocks_cd;
  int cd_index = threadIdx.x + (cd_block) * BlockDim.x;

  __shared__ double UP_sh[MAX_KAB], ... ;

  UP_sh[threadIdx.x] = UP_gl[ab_index + threadIdx.x*nab];
  ...
  for(unsigned int kcd = 0; kcd < contraction_cd; ++kcd){
    const double UQ = UQ_gl[cd_index+ncd*kcd];
    ...
      for(unsigned int kab = 0; kab < contraction_ab; ++kab){
        const double UP = UP_sh[kab];
        ...
          m[0] = boysf(..);
          ERI_array[ab_index*(stride)+ cd_index] += m[0]*theta;
   } }
}
```

```
#pragma omp target enter data map(to:h_Kab_array[0:size/sizeof(...)]);
...
kernel_0000( h_Kab_array, ..., ERI_array);
```

```
          void kernel_0000( int* __restrict__ Kab_gl, ... ,
                            double*  __restrict__ ERI_array ){
#pragma omp target teams distribute parallel for
 for(int i=0; i< nab*ncd; i++) {
  int ab_index = i % (ncd);
  int cd_index = i / (ncd);




  ...
  for(unsigned int kcd = 0; kcd < contraction_cd; ++kcd){
    const double UQ = UQ_gl[cd_index+ncd*kcd];
    ...
      for(unsigned int kab = 0; kab < contraction_ab; ++kab){
        const double UP = UP_gl[ab_index+kab*nab];
        ...
          m[0] = boysf(..);
          ERI_array[ab_index*(stride)+ cd_index] += m[0]*theta;
    } }
} }
```

Input: (ss|ss) , 200 water cluster, PC-seg-0



Hardware: dual socket Skylake Intel Xeon Gold 6152 CPU + 4 NVIDIA V100 SXM2 GPUs
IBM Power9 (2 Power9 CPU + 4 NVIDIA Volta V100 GPUs)

CUDA to DPC++

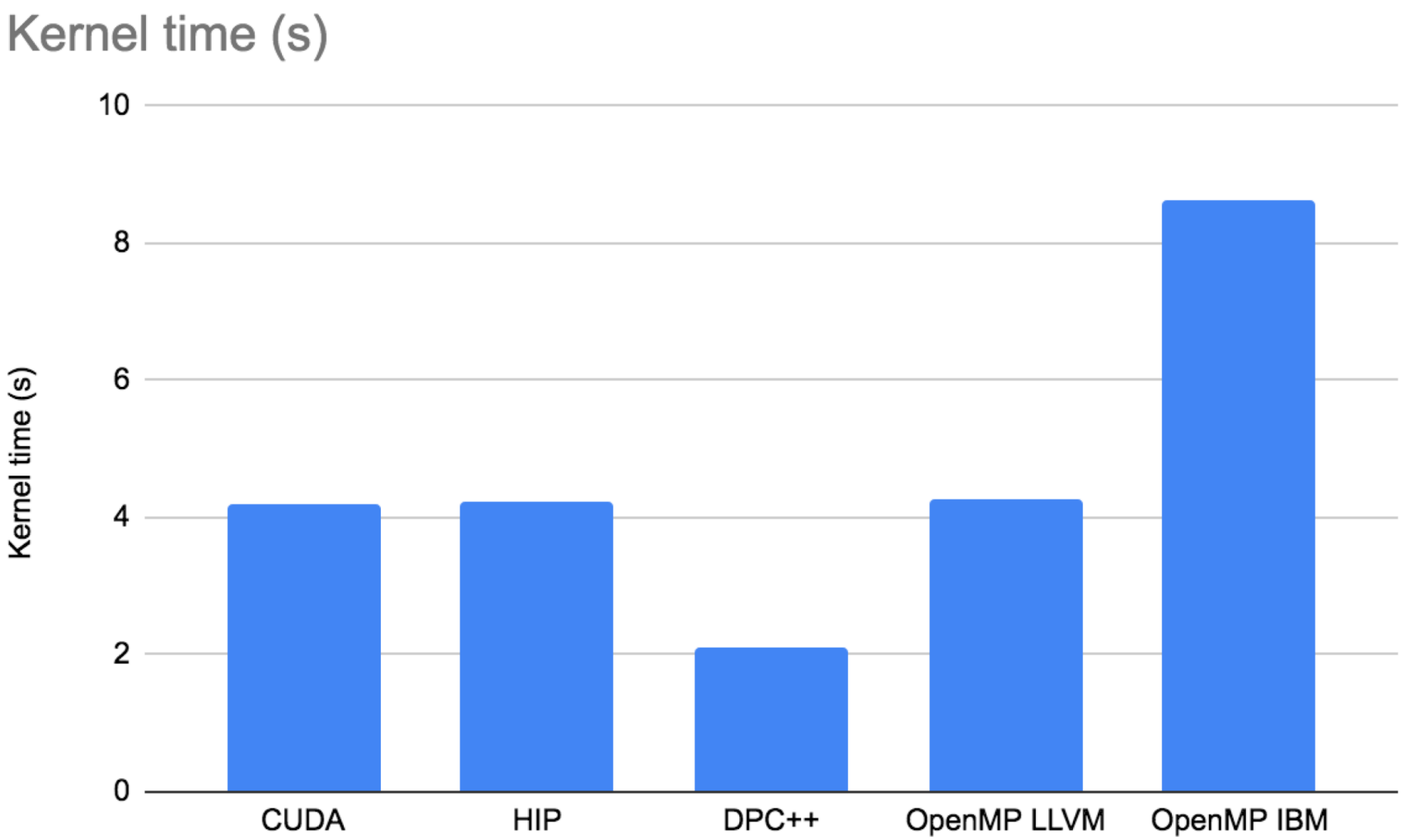
```
cudaMemcpy( d_Kab_array, h_Kab_array, size, cudaMemcpyDeviceToHost );
...
kernel_0000<<<grid_d,block_d>>>( d_Kab_array, ..., ERI_array);
```

```
__global__ void kernel_0000( int* __restrict__ Kab_gl, ... ,
                             ...,
                             double*  __restrict__ ERI_array ){
  int ab_index = blockIdx.x/n_blocks_cd;
  int cd_index = threadIdx.x + (cd_block) * BlockDim.x;

  __shared__ double UP_sh[MAX_KAB], ... ;

  UP_sh[threadIdx.x] = UP_gl[ab_index + threadIdx.x*nab];
  ...
  for(unsigned int kcd = 0; kcd < contraction_cd; ++kcd){
    const double UQ = UQ_gl[cd_index+ncd*kcd];
    ...
      for(unsigned int kab = 0; kab < contraction_ab; ++kab){
        const double UP = UP_sh[kab];
        ...
          m[0] = boysf(..);
          ERI_array[ab_index*(stride)+ cd_index] += m[0]*theta;
```

```
sycl::buffer<double, 1> Target(value, sycl::range<1>(stride*n_ab));
...
queue.submit([&] (sycl::handler& cgh) {
 auto target_acc = Target.get_access<sycl::access::mode::discard_write>(cgh);
    ...
    cgh.parallel_for<class kernel>(
            sycl::nd_range<1>(grid_d*block_d, block_d),
            [=] (sycl::nd_item<1> item) { kernel_0000( ... );
                                        });
              });
```

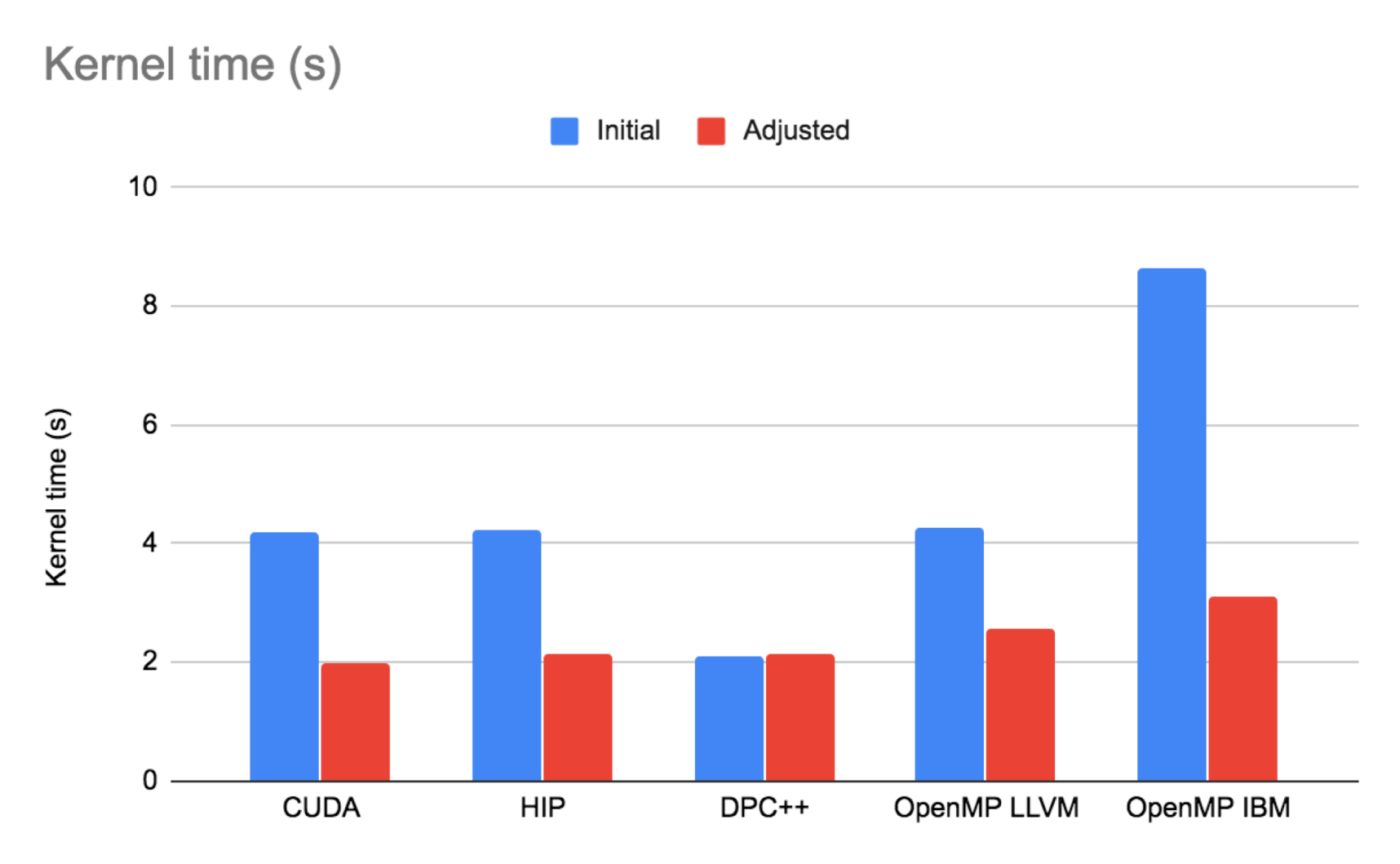
```
            void kernel_0000( int* __restrict__ Kab_gl, ... ,
                              double* UP_sh, ...
                              double*  __restrict__ ERI_array ){
   int ab_index = idx.get_group(0)/n_blocks_cd;
   int cd_index = idx.get_local_id(0) + (cd_block) * idx.get_local_range(0);



   UP_sh[idx.get_local_id(0)] = UP_gl[ab_index + idx.get_local_id(0) *nab];
   ...
   for(unsigned int kcd = 0; kcd < contraction_cd; ++kcd){
     const double UQ = UQ_gl[cd_index+ncd*kcd];
     ...
       for(unsigned int kab = 0; kab < contraction_ab; ++kab){
         const double UP = UP_sh[kab];
         ...
           m[0] = boysf(..);
           ERI_array[ab_index*(stride)+ cd_index] += m[0]*theta;
```

Input: (ss|ss) , 200 water cluster, PC-seg-0

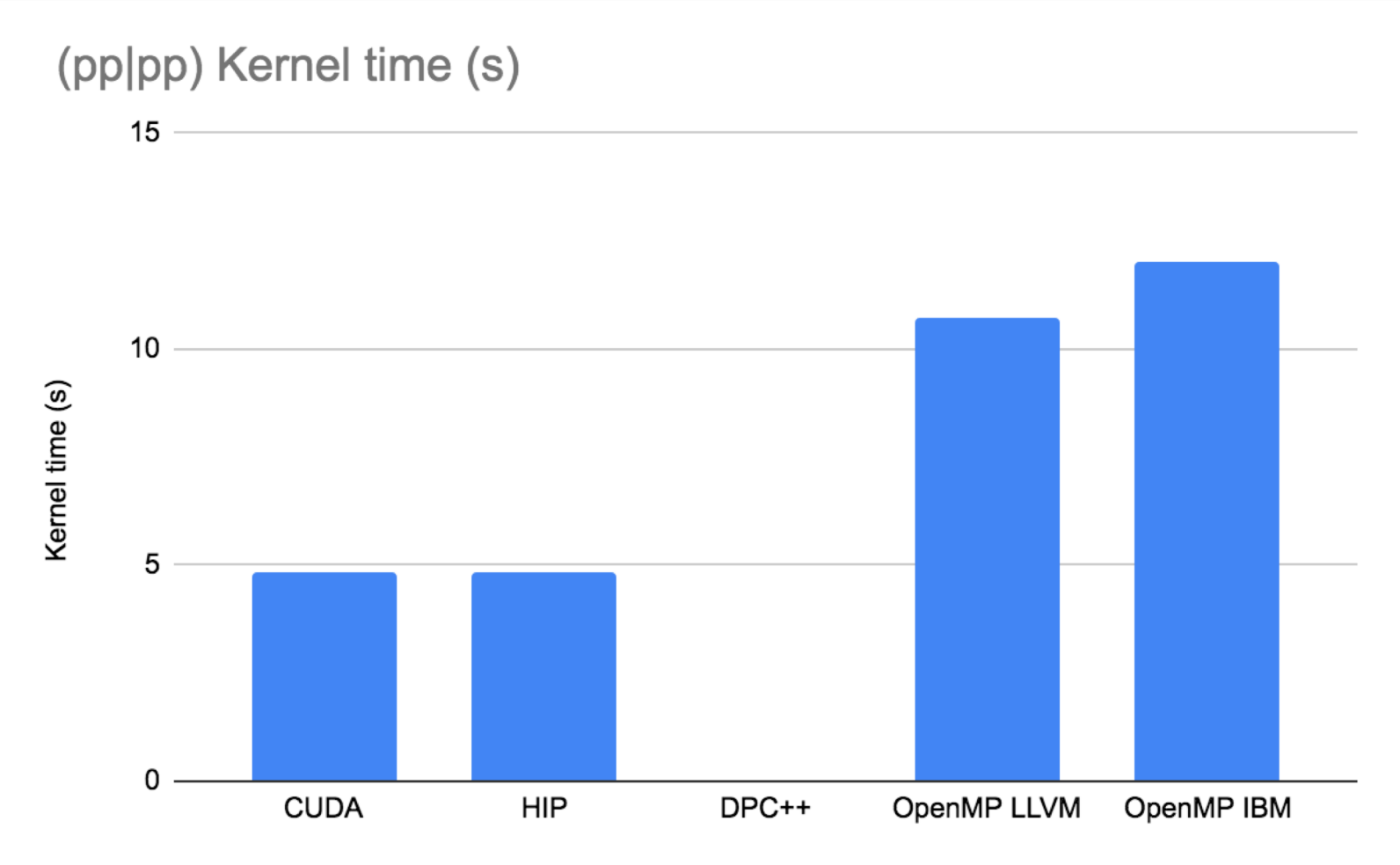


Hardware: dual socket Skylake Intel Xeon Gold 6152 CPU + 4 NVIDIA V100 SXM2 GPUs
IBM Power9 (2 Power9 CPU + 4 NVIDIA Volta V100 GPUs)

Code comparison: (ss|ss) timing



Code comparison: (pp|pp) timing



Summary and Next Steps

Summary

1. How difficult was it to switch the code from CUDA to other programming models?
   - For HIP, we used Hipify-perl, which worked well
   - Manually porting to OpenMP and DPC++ was doable
   - DPC++ was very verbose, but USM will probably help that
2. After changes, how does the performance compare?
   - Similar performance across programming models after modifications

Comments and next steps

- Can be hard to separate issues due to compilers and programming models
- Roofline analysis
- Goal is to finish porting code so that we can run on more hardware than Nvidia GPUs
  - Modifying the integral generator to generate code for other types of integrals
  - Porting the memory management part of the standalone code

Thanks!