

PERFORMANCE PORTABLE HETEROGENEOUS PROGRAMMING

Thomas Gorham

5/3/2022

Abstract

This report is intended to recount the learning experiences that occurred over the course of studying the concepts behind using Kokkos Performance Portability Framework to implement four HPC algorithms that execute in parallel. These algorithms are written in a hardware agnostic way, such that they are able to compile without code refactoring on manycore compute nodes with GPU accelerators. At the time of writing this, both the Kokkos Wiki and my wiki page/code base exist. Therefore, this report is designed to be more of an authentic discussion and summary, and less of a Kokkos dictionary or a copy/paste of the open source code that I implemented and is available publically. Instead, I aim to cover research concepts that went into this work, including: statistical data on the hardware behind Modern HPC Systems, the CPU and GPU software programming models for such systems, and how some of these Kokkos optimizations I implemented for the CPU, the GPU, and the architecture work under the hood. Anything I discuss in this summary report should be further explained and readily available in my code comments, my wiki page, or my video presentation. All of which are located in this Github repository.

This report is divided into three main sections:

Section A) The Purpose, Goal, and Significance of this work

Section B) Kokkos Necessities and Optimizations for heterogeneous computing

Section C) Semantics and Challenges of CPU GPU Programming with Kokkos

Section D) Most Notable Takeaways Confirmed By Programming and Research Analysis

Section A: The Purpose, Goal, and Significance of this work

The **purpose** of this work was to create a useful code base and informative wiki page to document my accumulated programming expertise on heterogeneous (CPU+GPU) architectures by utilizing Kokkos on a shared memory manycore node with multiple attached GPU accelerators. The **goal** in doing so is to show how Kokkos simplifies the challenges of programming heterogeneous architectures with programming abstractions that optimize data storage and parallel execution, and how different programming model back-ends can be set at compile time for maximum portability. The **significance** of this work is rooted in its overall alignment with current research regarding heterogeneous computing for performance portability. Below is a short table to briefly illustrate just a few Exascale Computing Project (ECP) research publications or technical reports that contain quite similar goals or findings to this work.

Aligning With Recent Research Directives		
Date	ECP Report/Publication	Description of Excerpt from the document
2/28/22	Publication: New Kokkos Release Improves Performance among exascale-era heterogeneous architectures [1]	"Writing and maintaining multiple application versions is impractical, and OpenMP, currently the only portable programming model supported by all vendors, is not only inconsistently implemented by different compilers, but the researchers believe that most C++ developers feel its pragma-based approach does not mix well with modern C++ software engineering practices."
2/11/22	App. Dev. Milestone Report [2]	Obtained speedup from Kokkos hierarchical parallelism and GPU shared memory
10/29/21	Helping Large-Scale Multiphysics Engineering And Scientific Applications Achieve Their Goals [3]	"Trillinos uses Kokkos Core." In fact, "Kokkos originally was designed to make Trilinos portable."

Section B: Kokkos Necessities and Optimizations

At the beginning of this project, I had previously built Kokkos on multiple architectures and written about some of the library’s basic capabilities. However, I hadn’t deeply explored or experimented with all of the Kokkos ecosystem features and optimizations. To aid in my understanding of how Kokkos simplifies heterogeneous computing challenges, I began my work by first reviewing the state of modern CPU-GPU architectures today, both from a hardware and software perspective. Statistical results from this part of my analysis are discussed in *Section 3*. Next, I reviewed the modern design implications of heterogeneous computations for *performance* and *portability*. With regards to *performance*, I then used Kokkos to optimize CPU/GPU memory access patterns and layouts both from compile time parameters and then controlling the default execution/memory spaces in the code, beginning with Program 1.

The code snippet below shows how these spaces can be explicitly controlled to set optimal access for the CPU or the GPU from within the source code.

Listing 1: Explicitly Optimizing the CPU

```
Kokkos::parallel_for("optCPU",
RangePolicy<Kokkos::HostSpace>
(start, end), [=] (const int i)
{
/* access data on cpu optimally
* Kokkos maps indices to cores
* in contiguous chunks and host
* space defaults to
* Kokkos::LayoutRight (row major)
*/
})
```

Listing 2: Explicitly Optimizing the GPU

```
Kokkos::parallel_for("optGPU",
RangePolicy<Kokkos::HIPSpace>
(start, end), [=] (const int i)
{
/* access data on an AMD GPU
* optimally. Kokkos maps indices
* to cores strided, and HIPSpace
* (or CUDASpace, etc) defaults to
* Kokkos::LayoutLeft (col major)
*/
})
```

When I say *explicitly controlled*, I’m referring to how I’m setting the execution space manually on line two in the examples above. For instance, *Kokkos::HostSpace* ensures the body of the code executes on the host device, where as *Kokkos::HIPSpace* ensures the body of the code executes on an AMD GPU accelerator with HIP.

IMPORTANT: If you don’t set the execution space manually as above, its set to the default execution space, based on what you compile for (such as serial CPU, parallel CPU, GPU kernel, etc). Kokkos default execution space is controlled by the CMake parameters at compile-time, and you can query the space in your source code via:

Listing 3: Execution Spaces

```
#include "Kokkos_Core.hpp"
#include <iostream>

int main(int argc, char** argv)
{
    Kokkos::initialize(argc, argv);

    std::cout << "\nDefault_execution_space:_ "
               << typeid(Kokkos::DefaultExecutionSpace).name() << "\n";
    std::cout << "\nDefault_Host_execution_space:_ "
               << typeid(Kokkos::DefaultHostExecutionSpace).name() << "\n";
}
Kokkos::finalize();
return 0;
}
```

The image below displays an example of some directions from my README.md, ultimately regarding how to let the compiler set these spaces automatically for maximum portability as I did in the majority of my code. This way, when porting your application to different machines, hard-coded execution spaces don't need to be changed.

For Building the Serial Backend:

```
mkdir build && cd build
cmake .. -DCMAKE_INSTALL_PREFIX=<Path --to-where-you-cloned-Kokkos-in-step-1>
        -DCMAKE_CXX_COMPILER=<path-to-your-g++>
```

For building with OpenMP Enabled

```
cmake .. -DCMAKE_INSTALL_PREFIX=<path-to-where-you-cloned-Kokkos-in-step-1>
        -DCMAKE_CXX_COMPILER=<path-to-your-g++>
        -DKokkos_ENABLE_OPENMP=ON
```

For building with CUDA Enabled

```
cmake .. -DCMAKE_INSTALL_PREFIX=<path-to-where-you-cloned-Kokkos-in-step-1>
        -DCMAKE_CXX_COMPILER=kokkos/bin/nvcc_wrapper
        -DKokkos_ENABLE_CUDA=ON
```

Keep in mind you should also pass the specific GPU architecture optimization at compile-time if compiling for the GPU and looking to obtain maximum performance. From my experiments, doing so yields a small but noticeable performance increase in the execution time of the code. On the Firefly Cluster at the SimCenter in Chattanooga, TN, this command to enable this optimization at compile-time is

$$-DKokkos_ARCH_VOLTA70 = ON$$

Additionally, there's optional runtime optimizations you can set for the CPU regarding the number of OpenMP threads and NUMA regions per the architecture. For code execution on Firefly node 2, I ran the Linux Bash shell command **lscpu** to determine the number logical CPUs on the machine. Again, from my experimentation, setting the threads to this max number yielded the most performance. For example, Firefly02 has 40 physical CPUs and 20 physical CPUs per socket. Of these 20 CPUs per socket, each has 2 threads. Thus, there are 80 Logical CPUs. So you can run with max threads in Kokkos by either

$$export OMP_NUM_THREADS = 80$$

One can also retrieve this number in code with OpenMP in the Kokkos environment using:

$$omp_get_max_threads();$$

The Kokkos library offers many specific optimizations, and the ones I utilize are reported in my Wiki. The final optimization I'd like to mention in this summary report is with regards to the machine's shared memory NUMA regions. From our **lscpu** output, we can identify the number of NUMA regions and tell Kokkos to optimize these regions accordingly. Since Firefly has two regions with 40 threads per region at run-time, higher performance on all programs in the repository can be achieved with the following run-time parameters appended:

`--kokkos- numa = 2 --kokkos- threads = 40`

I also recalled from a graduate level parallel computing class I took under Mr. Ryan Marshall, we had experimented with matrices access time in *C* programming language, and we found that one-dimensional row-major access was the fastest. Below is an image from his in-class experimentation to illustrate how one-dimensional row-major access is the fastest.

```

rmarshall@epyc:/home/rmarshall/cpsc5260/ipp-source/bad_ideas$ ./a1v2 1000
M[r*rowsize+c] secs: 0.0620765
M[r][c] secs: 0.0865936
M.at(r).at(c) secs: 0.0970541
rmarshall@epyc:/home/rmarshall/cpsc5260/ipp-source/bad_ideas$ ./a1v2 1000
M[r*rowsize+c] secs: 0.061913
M[r][c] secs: 0.100674
M.at(r).at(c) secs: 0.0966297
rmarshall@epyc:/home/rmarshall/cpsc5260/ipp-source/bad_ideas$ ./a1v2 5000
M[r*rowsize+c] secs: 1.60133
M[r][c] secs: 1.68513
M.at(r).at(c) secs: 2.96785
rmarshall@epyc:/home/rmarshall/cpsc5260/ipp-source/bad_ideas$ ./a1v2 10000
M[r*rowsize+c] secs: 6.69922
M[r][c] secs: 6.87273
M.at(r).at(c) secs: 11.9639
rmarshall@epyc:/home/rmarshall/cpsc5260/ipp-source/bad_ideas$ ./a1v2 10000
M[r*rowsize+c] secs: 6.69936
M[r][c] secs: 6.87469
M.at(r).at(c) secs: 12.1025
rmarshall@epyc:/home/rmarshall/cpsc5260/ipp-source/bad_ideas$

```

For this reason, I implemented a two-dimensional matrix in Program1 to compare the access time to my two-dimensional Kokkos::Views typedefs. I learned the difference between the two was negligible, confirming something I had learned in Listing 1 on page 2 of this report. Essentially, if you are filling a Kokkos::HostView, it defaults to contiguous row major chunks for optimal storage/access on the CPU.

Kokkos::Views are an extremely useful data storage abstractions with regards to application *portability*. Views are *N* dimensional abstractions that are able to store data in a different execution or memory space. Since I've written a paper about this before, I will spare the copy/paste definition and just report my most recent discovery that I validated with this project. When you create mirror views or do the deep copy for the GPU, Kokkos::Views also automatically optimize the layout for you. This is extremely beneficial in terms of reducing programming complexity, since you will only ever be printing from the CPU (in row major for matrices). The code below shows how a common HPC problem can be reduced to a hardware agnostic solution that can be configured for parallelism on the CPU and the GPU without code refactoring. Note, this is how I wrote my programs that utilize matrices, and because I use *Kokkos::Views* in the following example, this code also includes the optimal CPU/GPU memory access patterns *automatically*.

Listing 4: Tommy's Kokkos Template

```

// STEP1: DEFINE this is for vectors and matrices in disguise (evil laugh)
typedef Kokkos::View<double*> LinearType;

// USER DEFINED GPU KERNEL or PARALLEL CPU EXECUTION
void runKernel(LinearType::<ExecutionSpace> data, unsigned int N);

// ALWAYS PRINT ON HOST, so OK to hard code this function's exec space
void show(LinearType::HostMirror);

```

```

/* program entry point */
int main(int argc, char** argv)
{
    // STEP 3: INIT ENV
    Kokkos::initialize(argc, argv); /* initialize kokkos core */
    { /* start kokkos scope */
        // STEP 4: INIT DATA
        const unsigned int N = 1000;
        LinearType x("x", N);
        LinearType::HostMirror h_x = Kokkos::create_mirror_view( x );
        // Init LinearType vector x on host
        for (i = 0; i < N; ++i) { h_x(i) = 1; }
        // STEP 5: TELL THE GPU
        Kokkos::deep_copy(x, h_x);
        // STEP 6: RUN KERNEL
        runKernel(x, N);
        // STEP 7: COPY BACK TO CPU IF NEEDED
        Kokkos::deep_copy(h_x, x);
        // STEP 8: PRINT ON HOST
        show(x); /*function definition loop through x.extent(0) */

    } /* close kokkos scope */
    Kokkos::finalize(); /*free memory*/
}

```

Section C: Semantics and Challenges of CPU GPU Programming with Kokkos

To understand the implications of this project and what exactly Kokkos solves, let me start by saying High Performance Computing, or, HPC, means one thing:

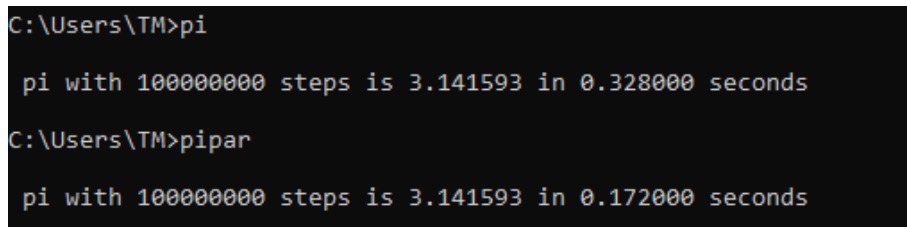
more compute resources = more computational power = the ability to solve problems faster.

Predominately, HPC enables us to solve difficult problems across many scientific fields that were once considered "out of reach". One example of a problem that we have solved thanks to developments in HPC was decoding the Human Genome [5].

But how are these problems solved?

By writing scientific applications that make use of more compute resources via parallel execution on HPC machines. [6] has a list of several hundred scientific applications managed by the National Institutes of Health, one of the many agencies where HPC is utilized.

What do we use to solve these problems with parallel execution?



```

C:\Users\TM>pi
pi with 100000000 steps is 3.141593 in 0.328000 seconds
C:\Users\TM>pipar
pi with 100000000 steps is 3.141593 in 0.172000 seconds

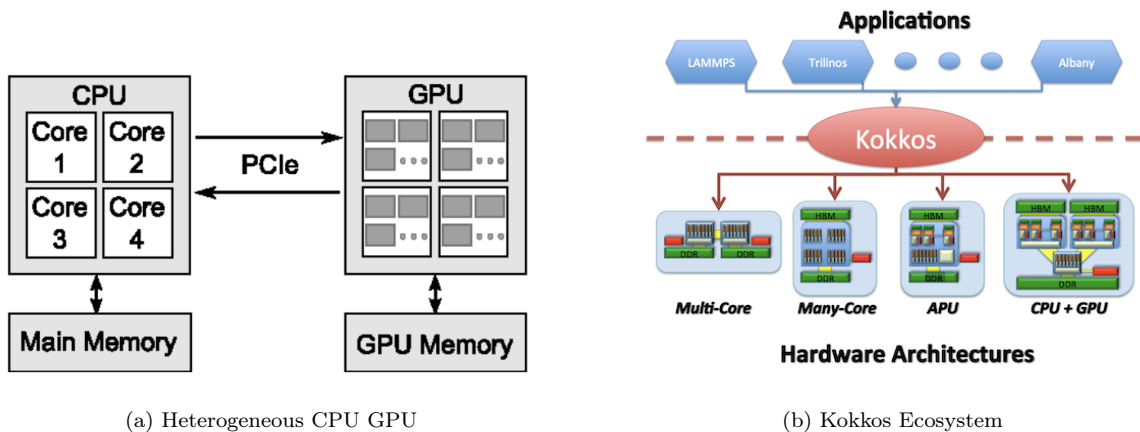
```

(a) 1 to 2 thread speed up with OpenMP

In the past, HPC applications were mostly written in Fortran. However, recent studies like [7] have revealed that the desired features of programming languages for data-intensive HPC applications are *portability*, *performance* and *usability*. In this study, the authors indicate that modern abstractions in C++ ultimately alleviate some of the issues regarding the complexity of writing performance driven HPC applications. Thus, C++ has become the more sought after language of choice for HPC. Furthermore, the latest versions of Parallel Programming APIs (e.g., OpenMP, CUDA, MPI) and Performance Portability Frameworks (e.g., Kokkos, RAJA, DPC++) build their code around C/C++ because the vast majority of new HPC application code is written in modern C++ [7,8]. C++ language features such as templates and lambdas are important features for performance, and lambdas are the key to dispatching parallel work in Kokkos.

What is Kokkos for then? Challenge #1

As the complexity and diversity of HPC machines increases in order to generate more computational power, so does the difficulty of programming reusable solutions for optimal performance. On heterogeneous (CPU-GPU) machines that contain at least one GPU accelerator, this means that HPC applications and parallel programming libraries must exploit increasingly finer levels of parallelism within their codes to sustain scalability on both of these devices (the CPU and the GPU). For example, it's common practice in this sort of environment to offload expensive computations to vectorized accelerators such as the GPU as a means for achieving fine-grained parallelism. One particular problem from this data offload from the CPU (the host) to the GPU (the device) is optimizing memory access patterns. This is because these two computational resources store and process data quite differently, and the data structure layouts in GPU memory often lead to sub-optimal performance for programs designed with a CPU memory interface. In other words, application performance is highly sensitive to irregularity in memory access patterns. Though Contemporary portable programming models address manycore parallelism (e.g., OpenMP, OpenACC, OpenCL), they don't yet address diverse and conflicting set of constraints on memory access patterns across devices.



SOLUTION #1: Kokkos solves this memory access problem by unifying abstractions for both fine-grain data parallelism and memory access patterns

What is Kokkos for then? Challenge #2

Another particular problem of programming diverse heterogeneous environments lies in something called **code portability**. For example, numerous programming models have been built to target select execution resources (e.g., the CPU/GPU) for specific HPC machine architectures so that the code can optimally make use of these resources. Since there's not really a standardized way of writing heterogeneous code that can overcome data transfer constraints, **it's not quite clear how performance portable a program written for an AMD-based machine will be to another (e.g., an Intel-based machine)**. What is clear

is that if you use the standard programming model for a heterogeneous NVIDIA-based machine and you want to run on an AMD-based machine, you will need to refactor your code to support AMD's programming model. As mentioned earlier, though this may only consist of switching to AMD's OpenMP5 specification for the CPU, the GPU code will need to be rewritten in HIP. In other words, changing vendor-standardized programming models to run on different machines requires code to be rewritten per the specifications of the machine's respective programming model. Ultimately, this becomes a particular problem for large-scale HPC applications that on average consist of 300,000 - 600,000 lines of code. In other words, rewriting code is expensive.

SOLUTION #2: Kokkos solves this code portability problem across all modern HPC architectures.

Essentially, the Kokkos API allows developers to express parallelism that is able to compile and execute:

- In Serial on the CPU
- In Parallel on the CPU
- In Parallel on the CPU and GPU

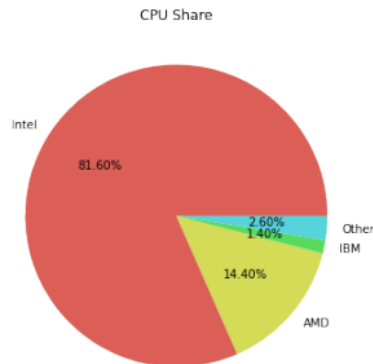
Expressing parallelism in this way enables developers to avoid rewriting thousands of lines of code with the specific API that is tied to the manufacturer of either chip. Thus, this removes the need to refactor the source code itself, and enables HPC applications to run on theoretically any HPC machine with ease.

Section D: Most Notable Takeaways From Completing this Project

1. **The uncertainty of *performance portability* in HPC is rooted in the lack of software uniformity and hardware heterogeneity.** Consider the following standards for GPU programming models that are restricted to programming GPUs based upon the manufacturer of the GPU chip itself (e.g., NVIDIA, AMD, Intel, etc.). Additionally, these programming models are tied to their vendor-specific version of OpenMP5 for parallelism on the CPU.

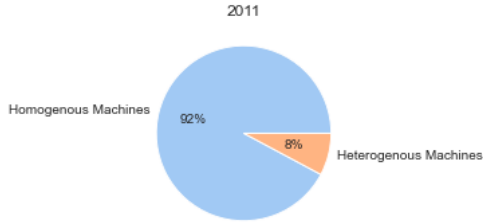
Understanding a Heterogeneous Environment		
Standard GPU Prog. Model	GPU Vendor	OpenMP Specification
CUDA	NVIDIA	NVIDIA's OpenMP
HIP	AMD	AMD's OpenMP
OpenCL or DPC++	Intel	Intel's OpenMP

Note: To maximize CPU/GPU code portability on the fastest HPC Architectures without a Framework like Kokkos, choose Intel's OpenMP5 and NVIDIA's CUDA for expressing parallel execution within the application. This will cover the most bases, as the fastest hardware architectures in the world overwhelmingly use NVIDIA GPU accelerators and Intel CPUs (particularly Xeon). Below is a visualization I generated with Python after analyzing the CPU share from the Top500's November 2021 dataset

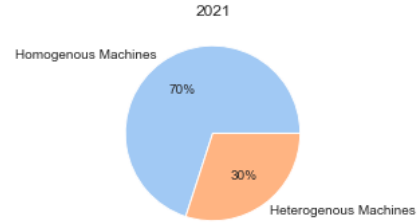


2. **The increase of Heterogeneity in HPC Machines can be confirmed by data:** Part of the significance of this work is the reality of modern architectures of HPC computers and clusters. From my analysis of the Top500's official list released in November of 2021, I found that even though CPU-based machines still control 70% of the field over those with an attached accelerator, this is a 275% increase over the last 10 years. Below is the equation I used to determine this percent of increase, followed by two pie charts I generated in Python to visualize it.

$$\% \text{ increase} = \frac{.3 - .08}{.08} * 100$$



(a) Fastest machines in the world, 10 years ago



(b) Fastest machines in the world, half a year ago

Homogeneous Intel Xeon Machine's Currently Own The Real Estate

3. **It's best to know Linux for Modern CPU GPU Programming:** According to the data, the most powerful supercomputers in the world (based on the LINPACK benchmark) overwhelmingly use of some flavor of Linux OS. No other operating system is present, nor has been for a long time (if ever).
4. **High CPU core count is important for achieving high performance in Linear Algebra Applications:** Not only did I achieve best performance when I set Kokkos to use the maximum amount of logical cores on Firefly Node 2, the total number of cores in the machine also have strong correlation *about 75%* with LINPACK performance and theoretical peak performance of these top-performing machines. From this combined knowledge, it seems appropriate to make use of all cores or optiize shared memory NUMA regions when designing parallel linear algebra algorithms for execution on the CPU.
5. **GPU offload should be the main parallel execution target when:** Implementing embarrassingly parallel programs and those that have a large number of mathematical computations. From my experimentation, exploiting parallelism from both the CPU and the GPU yields the most speedup in these cases. Recall that when using Kokkos, the *execution space* determines this execution target, and that it hierarchically trickles down from:

$$Parallel \ GPU \ Execution \rightarrow Parallel \ CPU \ Execution \rightarrow Serial \ CPU$$

when using CMake parameters to configure this.

6. **In some cases, GPU Performance falls flat with Kokkos:** This was seen with the Kokkos implementation of `std::sort` in Program4. From some papers that I reviewed this semester regarding parallel merge sort, it may be best to sort in serial once the N , or, total problem size is small enough to mitigate this.
7. **Judiciously set Kokkos::ExecutionSpace and Kokkos::MemorySpace explicitly:** As opposed to the default which is set at compile-time, I have learned that if you are writing an application that you know will need to utilize one or many GPU accelerator(s), it can be beneficial to go ahead and hard code data that you know will explicitly need to be initialized for the GPU(s). In doing so, you

essentially optimize the memory access pattern for this data separately which can enable you to avoid the initial Kokkos::DeepCopy that is required to execute the Kokkos::View on the GPU. An example of this is in Listing 1 on page 2 of this report.

8. **Aside: Fortran is not always faster than C++** An HPC application from [2] that ported to C++ Cabana/Kokkos is 28% faster than the prior implementation which was in Fortran Cabana/Kokkos. This was on page 79 of [2] and I just found it interesting as I discussed why C++ was used for this project at the top of page 6.

The remainder of this paper is the performance results from my programs and the bibliography.
Thank you for reading.

Performance Results of Program 1

HPC Program 1 Algorithm: Matrix Multiplication	Serial Run	Kokkos Serial Run	Avg CPU Run	Avg GPU Run	CPU Speedup	GPU Speedup
Execution time measured in milliseconds on Firefly node 2	547.762	362.205	21.973	.002-5.667 (avg 1.2)	24.92886725	456.4683333
				GPU Util 80%		
				From others		
				running code		
				at the same time.		
Data Type: Double						
Equation: $y=Ax$						
Size of Data Structures (N): 10,000						
Size of Result (NxN): 100,000,000						

Performance Results of Program 2

HPC Program 2 Algorithm: Conway's GOL	Serial Run	Kokkos Serial Run	Avg CPU Run	Avg GPU Run	CPU Speedup	GPU Speedup
Execution time measured in milliseconds on Firefly node 2	1767.11	2950.41	153.956	6.193	19.16398192	476.4104634
Size of Data Structures (N): 1024						
Generations: 100						

Performance Results of Program 3 (using Python)

```
[4]: df2 = df[['Execution Time (sec)', 'Programming models enabled', 'speedup']]
```

```
[7]: df2.iloc[:4]
```

	Execution Time (sec)	Programming models enabled	speedup
0	0.000469	openmp, cuda	2523.500672
1	1.182790	serial	NaN
2	0.269459	openmp	4.389499
3	0.002201	cuda	537.299669

Performance Results of Program 4

HPC Program 4 Algorithm: Parallel Merge Sort					
	Kokkos Serial Run	Avg CPU Run	Avg GPU Run	CPU Speedup	GPU Speedup
Execution time measured in milliseconds on Firefly node 2	424.813	23.304	147.006	18.22918812	2.889766404

References

- [1] ECP "New Kokkos Release Improves Performance Portability Among Exascale-Era Heterogenous Architectures"
- [2] ECP "2021 Application Development Milestone Report"
- [3] ECP "Helping Large-Scale Multi-Physics Engineering and Scientific Applications Achieve Their Goals"
- [4] ECP Nov. 19, 2020 "ECP Software Technology Capability Assessment Report Public"
- [5] National Library of Medicine "Decoding the human genome"
- [6] BLOWULF High Performance Computing At The NIH "Scientific Applications on NIH HPC Systems"
- [7] Science Direct "Programming languages for data-Intensive HPC applications: A systematic mapping study"
- [8] International Journal of Innovative Science and Research Technology "Programming Language for Data Intensive HPC Applications: Applications and its Relevance"