1 SYCL Memory Management

Yojan Chitkara
October 11, 2023

2 Introduction

This text describes the different techniques implemented in managing memory in SYCL. We look at 2 kernel implementations *Matrix Addition* and *Matrix Multiplication* each of which are managing memory in 3 different ways. We start off with describing what the different memory management techniques are, followed by our Kernel implementations using each memory management technique. With this we describe the results obtained on each of our kernel implementations. Finally we summarize with what we observed was the most ideal implementation for the kernels described.

2.1 Memory Management

SYCL enables memory management for user applications through two basic mechanisms.

1) ***Malloc*** *pointer manipulation through standard C++ calls (Often referred to as USM or Unified Shared Memory)*

Unified Shared Memory (USM) is a pointer-based memory management in SYCL. USM is a pointer-based approach that should be familiar to C and C++ programmers who use malloc or new to allocate data. USM simplifies development for the programmer when porting existing C/C++ code to SYCL. With USM, the developer can reference that same memory object in host and device code.

Types of USM

Type	Function Call	*Description*	Accessible on Host	Accessible on Device
Device	malloc_device	Allocation on Device (explicit)	NO	YES
Host	malloc_host	Allocation on Host (implicit)	YES	NO
Shared	malloc_shared	Allocation can migrate between host and device (implicit)	YES	YES

2) ***Buffers*** *that encapsulate data and can be accessed using "Host/Device Accessors"*

Device and host can either share physical memory or have distinct memories. When the memories are distinct, offloading computation requires copying data between host and device. SYCL does not require the programmer to manage the data copies. By creating Buffers and Accessors, SYCL ensures that the data is available to host and device without any programmer effort. SYCL also allows the programmer explicit control over data movement when it is necessary to achieve best peformance.

2.2 Kernel Implementations

The user application implements Matrix Multiplication kernels in SYCL for two underlying hardware - GPU and CPU. For the purpose of this Section, assume 3 Matrices - Matrix A and Matrix B (NxK and KxM - 8192xK and Kx8192) and Matrix C (NxM = 8192x8192)

Implementation 1 - Allocate memory for A and B on Host and Device using malloc (for host) and malloc_device (for device).

```
60      // Initialize Vectors and Print Values
61      double *H_a = static_cast<double*>(malloc(N*K*sizeof(double)));
62      double *H_b = static_cast<double*>(malloc(K*M*sizeof(double)));
63      double *H_c = static_cast<double*>(malloc(N*M*sizeof(double)));

91      // Initialize Device Mem and copy host data over to device for processing
92      auto *D_a = static_cast<double*>(malloc_device<double>(N*K,q));
93      auto *D_b = static_cast<double*>(malloc_device<double>(K*M,q));
94      auto *D_c = static_cast<double*>(malloc_device<double>(N*M,q));
```

Initialise the two input matrices A and B on the host and copy the data over to the device for computation.

```
104         auto e1 = q.memcpy(D_a,H_a,(sizeof(double)*N*K));
105         auto e2 = q.memcpy(D_b,H_b,(sizeof(double)*K*M));
```

The Kernel works on the copied device data using a "parallel_for" reduction.

```
107         // Kernel to multiply the two Two-Dim Vectors
108
109         q.parallel_for(range<2>(N,M), {e1,e2}, [=](auto index){
110
111             int row = index.get_id(0);
112             int col = index.get_id(1);
113
114             double sum = 0.0;
115             for(int k=0;k<K;k++)
116                 sum += D_a[row*K + k] * D_b[k*M + col];
117
118             D_c[row*N + col] = sum;
```

Finally the result of the computation is copied back to the host.

```
132         q.memcpy(H_c,D_c,sizeof(double)*N*M,e4).wait();
```

Implementation 2 - Allocate memory for A and B on Host and Device using malloc_shared.

```
61      // Initialize Vectors and Print Values
62      double *Mat_A = static_cast<double*>(malloc_shared(N*K*sizeof(double),q));
63      double *Mat_B = static_cast<double*>(malloc_shared(K*M*sizeof(double),q));
64      double *Mat_C = static_cast<double*>(malloc_shared(N*M*sizeof(double),q));
```

Initialise the two input matrices A and B and initiate the kernel on the device for computation. In this case, there is no need for explicit copy of data to or from the device to initiate computation or after the result is computed.

```
 97         // Kernel to multiply the two Two-Dim Vectors
 98
 99         q.parallel_for(range<2>(N,M), [=](auto index){
100
101             int row = index.get_id(0);
102             int col = index.get_id(1);
103
104             double sum = 0.0;
105             for(int k=0;k<K;k++)
106                 sum += Mat_A[row*K + k] * Mat_B[k*M + col];
107
108             Mat_C[row*N + col] = sum;
```

Implementation 3 - Allocate memory for A and B on Host using standard data structures (Array, Vector,etc).

```
 62     // Initialize Vectors and Print Values
 63     std::vector<double> Mat_A(N*K,10.0);
 64     std::vector<double> Mat_B(K*M,20.0);
 65     std::vector<double> Mat_C(N*M,0.0);
```

Initialise SYCL "Buffers" for each Matrix data structure.

```
 85     buffer<double,2> Buf_a(Mat_A.data(),range<2>(N,K));
 86     buffer<double,2> Buf_b(Mat_B.data(),range<2>(K,M));
 87     buffer<double,2> Buf_c(Mat_C.data(),range<2>(N,M));
```

These buffers can now be accessed using "Host and Device Accessors" in the kernels (Device Accessor)

```
 99         q.submit([&] (handler &h)
100         {
101             accessor D_a(Buf_a,h);
102             accessor D_b(Buf_b,h);
103             accessor D_c(Buf_c,h);
```

and outside the kernels (Host Accessors).

```
143     host_accessor result(Buf_c,read_only);
```

The Kernel works on the device accessors to compute the Matrix Product.

```
 98         // Kernel to add the two Two-Dim Vectors
 99         q.submit([&] (handler &h)
100         {
101             accessor D_a(Buf_a,h);
102             accessor D_b(Buf_b,h);
103             accessor D_c(Buf_c,h);
104
105             h.parallel_for(range<2>(N,M), [=](auto index){
106                 int row = index.get_id(0);
107                 int col = index.get_id(1);
108
109                 double sum = 0.0;
```

```
110                 for(int k=0;k<K;k++)
111                     sum += D_a[row][k] * D_b[k][col];
112
113                 D_c[row][col] = sum;
```

2.3 Hardware Selector

SYCL allows mechanisms to switch between the underlying hardware on which the kernels are executed without significantly altering the kernel implementation.
To select CPU

```
58      queue q(cpu_selector_v);
```

To select GPU

```
58      queue q(gpu_selector_v);
```

3 Results

The results were derived on Frontera with the following hardware specification.
CPU : Intel(R) Xeon(R) Platinum 8480+
GPU : Intel(R) Data Center GPU Max 1100 (Matrix Multiplication)
We have summarised the results below for the two hardware implementations.

3.1 GEMM

This kernel performs SGEMM (single precision) and DGEMM (double precision) matrix-matrix products for different values of K. Using this, we change the arithmetic intensity of the workload, thereby increasing it's compute demands. The memory requirements of this workload scale as $O(n^2)$ and the compute demands are of the order $O(n^3)$. Hence, for large matrix sizes, this benchmark is compute intensive. Our results indicate for lower sizes of K, buffers take up a significant chunk of time compared to double copy and malloc_shared. This happens particularly due to fixed setup time for buffers (initialisation of accessor pointers) that dominates execution time for low sizes of K. As arithmetic intensity increases, we observe execution time become dominated by computation and for K = 2048, we observe all three memory management techniques take roughly the same amount of time. As K increases beyond 2048 and workload approaches maximum compute intensity, we observe the benefits of the Buffer memory management technique to schedule data transfers during bursts of compute thereby minimising execution time compared to malloc shared and double copy. With this, we observe an overall speedup of 30x for SGEMM and 35x for DGEMM (average GEMM speedup $\sim$ 33x) with the baseline estimated speedup for compute workloads at 34.5x.

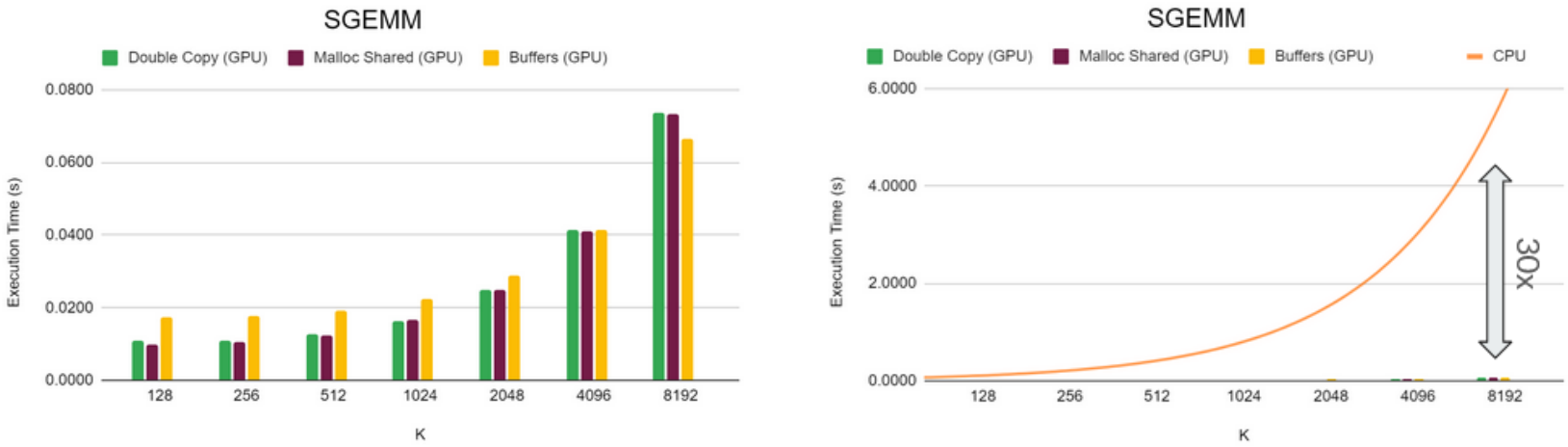

Figure 1: SGEMM Speedup on GPU

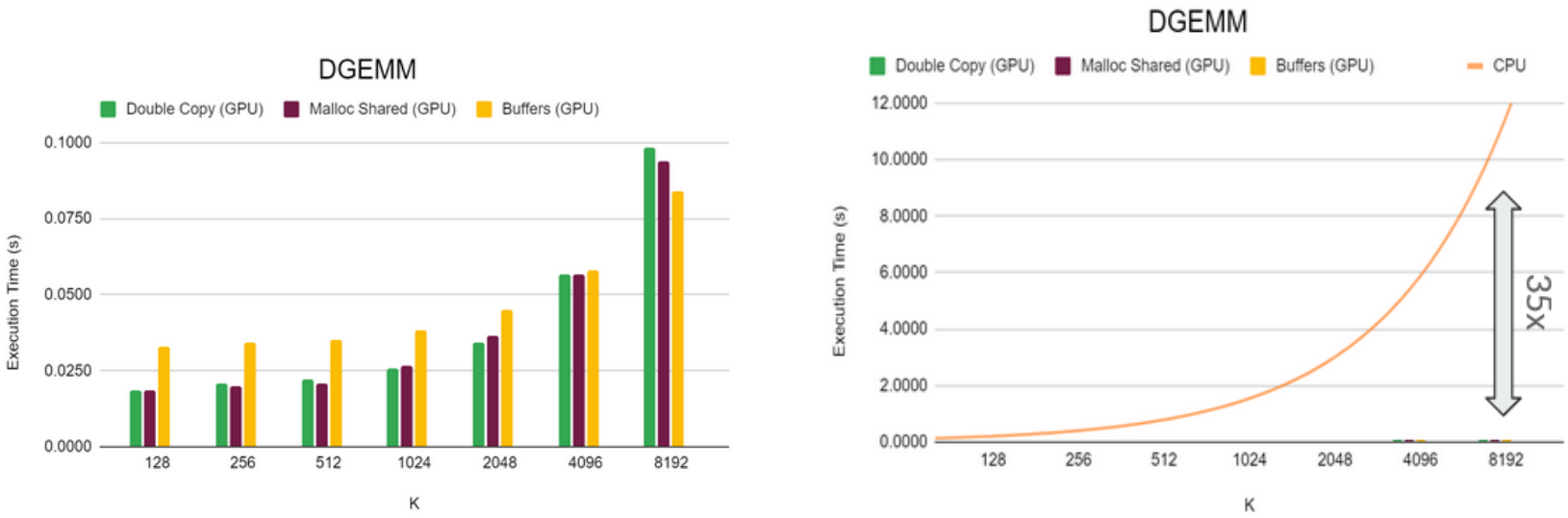

Figure 2: DGEMM Speedup on GPU

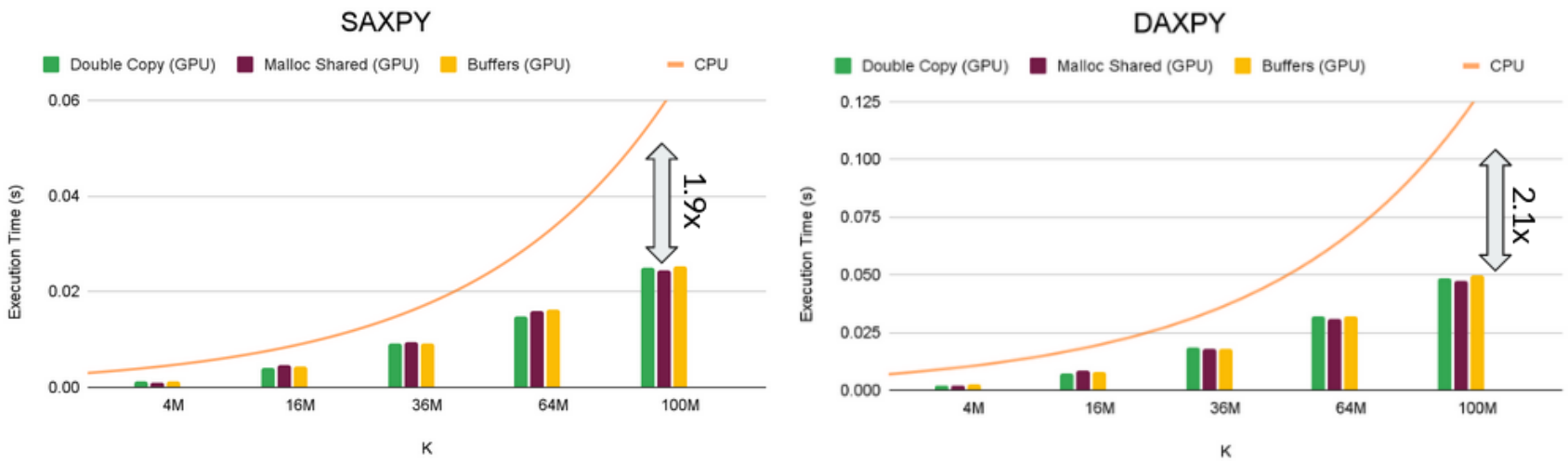

Figure 3: AXPY Speedup on GPU

3.2 AXPY

This kernel performs SAXPY (single precision) and DAXPY (double precision) vector-vector dot products for different vector lengths. We vary the length of vectors to change the memory demands of the workload. AXPY being a memory bound workload, would scale in execution time with an increase in vector length needed for the dot product. We observe that these workloads are dominated by internal memory transfers as opposed to data transfer overhead from the CPU to the GPU. It is for this particular reason that any change in data transfer technique minimally affect changes in execution time. The average observed speedup is $\sim$ 1.9x for SAXPY and $\sim$ 2.1x for DAXPY (average AXPY speedup of 2x) with the baseline expected speedup for memory workloads at 3.8x.

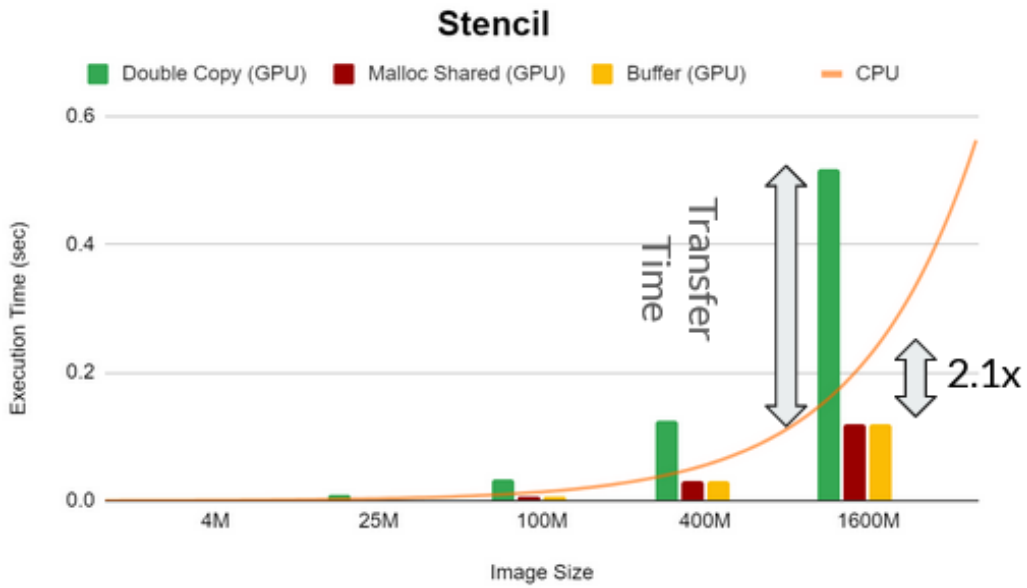

Figure 4: Stencil Speedup on GPU

3.3 STENCIL

This kernel performs convolution of a K*K image with a fixed 3*3 kernel matrix. As the size of the filter is small, the ratio of memory operations and compute operations is O(1). Hence, this is a memory intensive workload. On average we observe a speedup of 2.1X as shown in Fig. 4. Double copy has the worst performance due to a high transfer overhead that cannot be masked. Using malloc shared and buffers for memory management helps in hiding transfer latency by fetching blocks from memory on demand whenever required.

3.4 MANDELBROT SET

This kernel generates the Mandelbrot set fractal image. Each pixel is rendered independently and given the nature of the fractal image, numerous iterations of rendering are essential. Each iteration is an input to the next, resulting in a high quality image as the output. We only found an implementation of this workload which used buffers for memory management. The results for this benchmark are shown in Fig 6. As can be observed, the GPU has a 4X better performance when compared to the CPU which suggests that this is a memory intensive application.

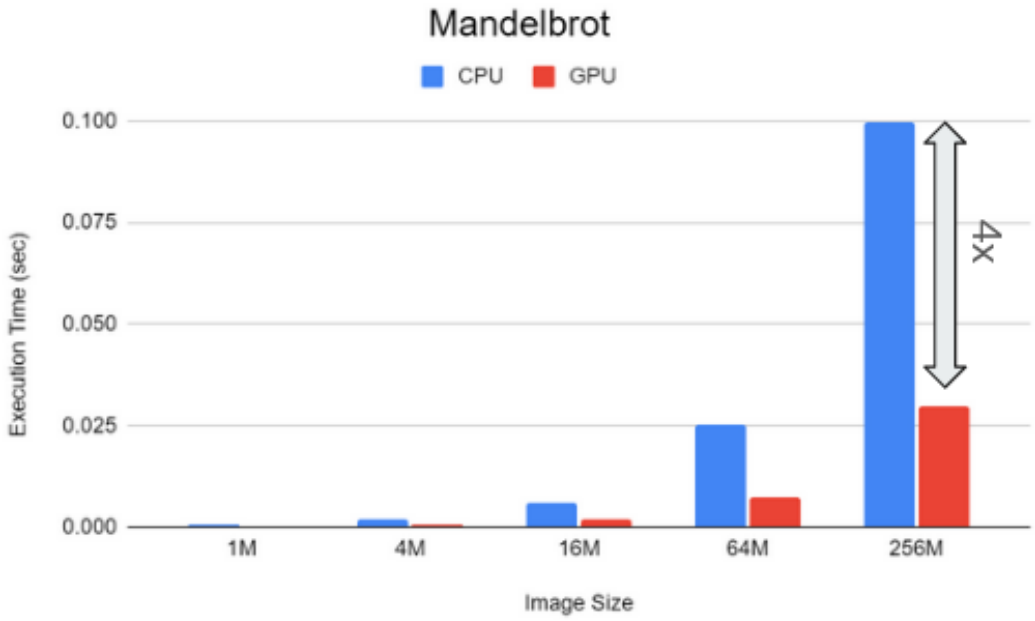

Figure 5: Mandelbrot Speedup on GPU

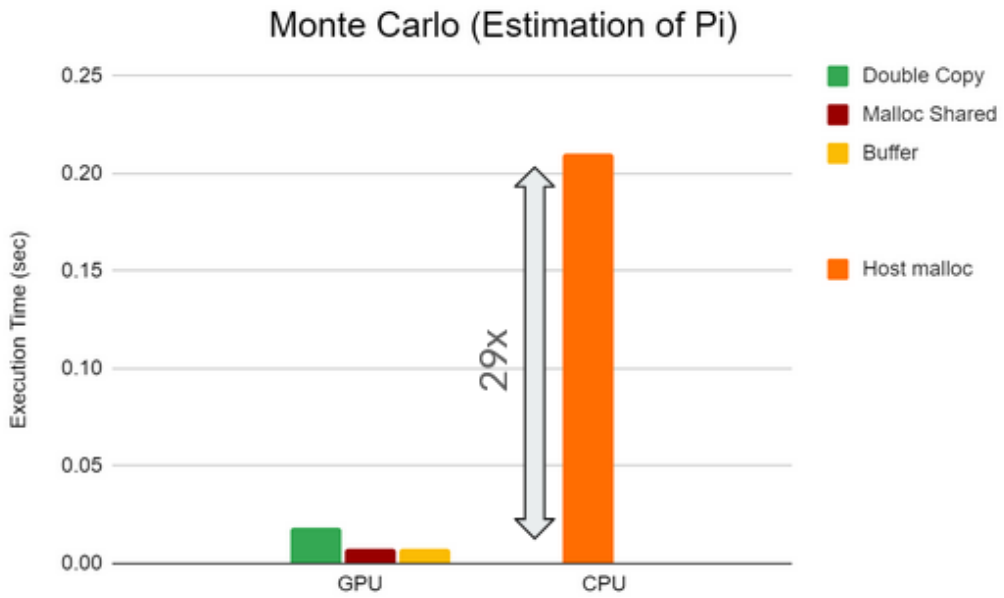

Figure 6: Monte-Carlo Speedup on GPU

3.5 MONTE CARLO

Monte Carlo workloads are a set of benchmarks which rely on repeated random sampling to obtain numerical results. Since the underlying concept relies on accuracy of computation for estimating the value of pi, we are able to classify this workload as compute intensive. This implementation is available for all three memory management techniques and we were able to observe a speedup of 30x on a GPU over a CPU implementation of the same.

4 Summary

The cumulative results for all benchmarks are shown in Fig. 7. On average compute intensive benchmarks show a speedup of 30-35x and memory intensive applications show a speedup of 2-4x. These numbers align with our performance estimation using baseline specifications of the two machines. Additionally, we observe that memory management using buffers on SYCL runtime performs better than all other memory management techniques. Among these, malloc shared (implicit copying on demand) performs better than double copy, which tends to incur high transfer overhead in some cases.

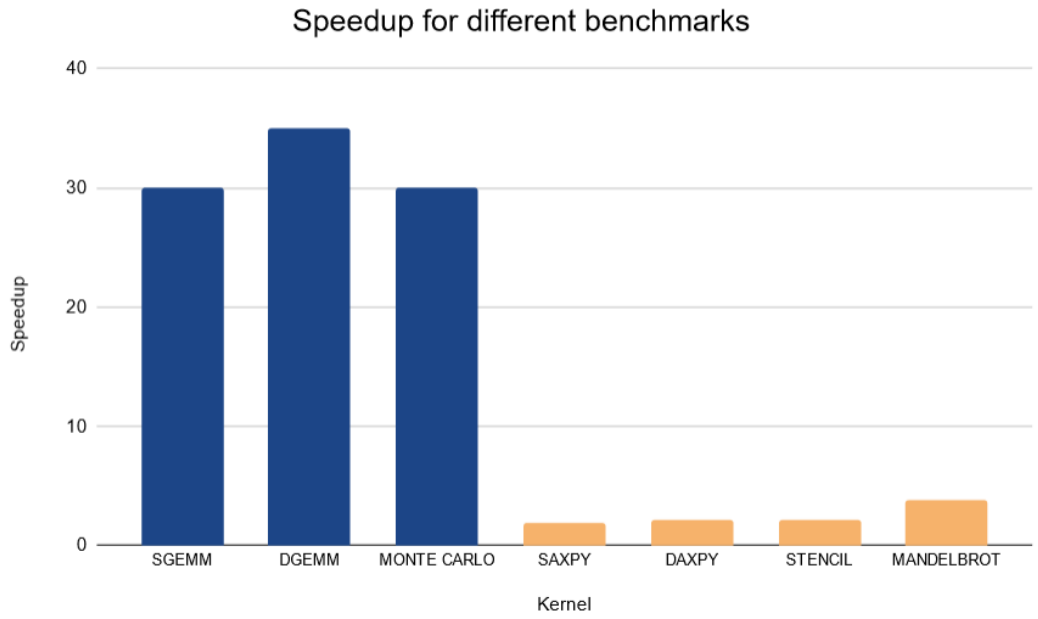

Figure 7: Overall speedup per kernel on GPU