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 CPU

3.1.1 Matrix Multiplication

Implementation 1 - The first implementation with Input Matrices A (8192xK) and B (Kx8192) with varying K from 64 to 4096 displays a monotonically rising graphs with an inflection at points K=128 and K=256.

N	M	K	TTC (Sec)
8192	8192	64	0.4002
8192	8192	128	0.4288
8192	8192	256	0.3557
8192	8192	512	0.3588
8192	8192	1024	0.3983
8192	8192	2048	1.4838
8192	8192	4096	4.1014

Implementation 2 - The second implementation with Input Matrices A (8192xK) and B (Kx8192) with varying K from 64 to 4096 displays a monotonically rising graphs with an inflection points K=128,K=256 and K=512 and K=1024

N	M	K	TTC (Sec)
8192	8192	64	0.3907
8192	8192	128	0.4135
8192	8192	256	0.3526
8192	8192	512	0.3414
8192	8192	1024	0.3654
8192	8192	2048	1.4513
8192	8192	4096	4.5829

Implementation 3 - The third implementation with Input Matrices A (8192xK) and B (Kx8192) with varying K from 64 to 4096 displays a monotonically rising graphs with an inflection at points K=128 and K=256.

N	M	K	TTC (Sec)
8192	8192	64	0.3531
8192	8192	128	0.3961
8192	8192	256	0.3423
8192	8192	512	0.3906
8192	8192	1024	0.5317
8192	8192	2048	2.4641
8192	8192	4096	5.9775

3.1.2 Summary

We notice that for K < 256, SYCL Buffers perform significantly better than the traditional malloc_* implementations. But as K becomes increasingly larger, the double copy with separate malloc_* on host and device offers significant performance improvements.

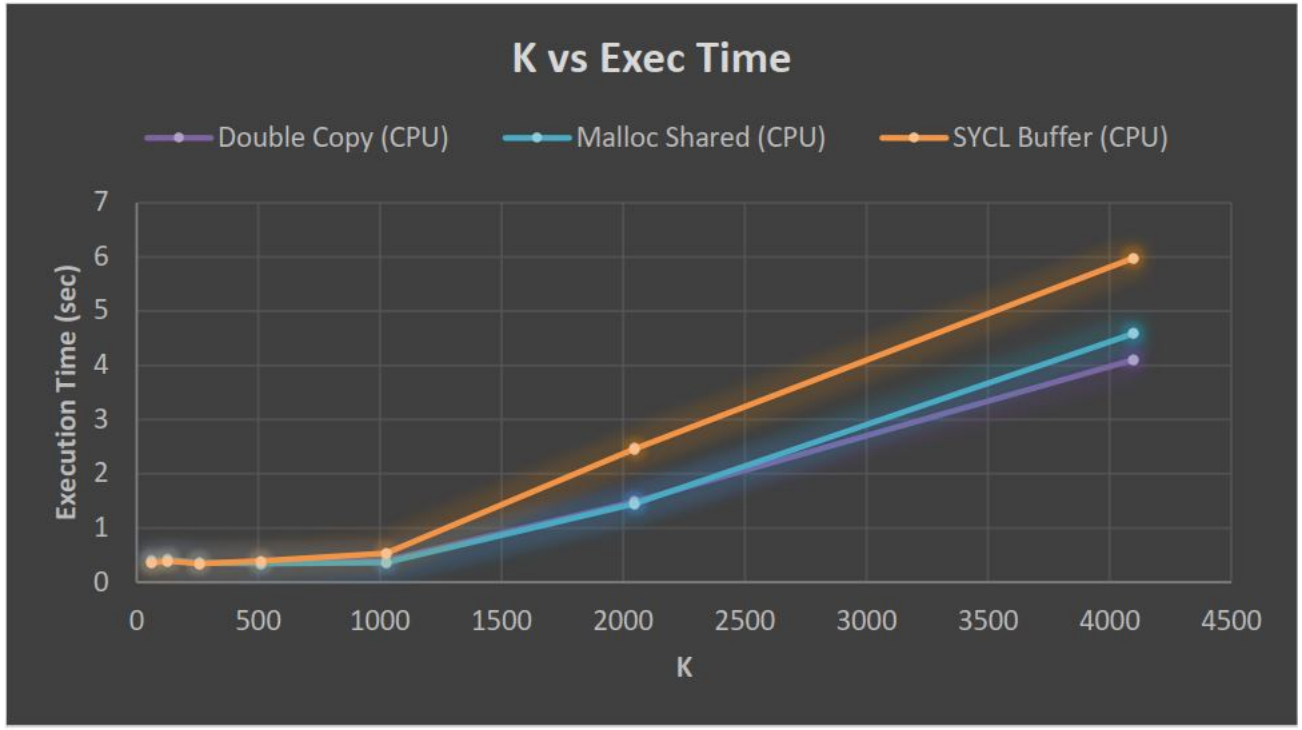

Figure 1: Compute Scaling on a CPU (K in MxNxK) across 3 different memory management techniques

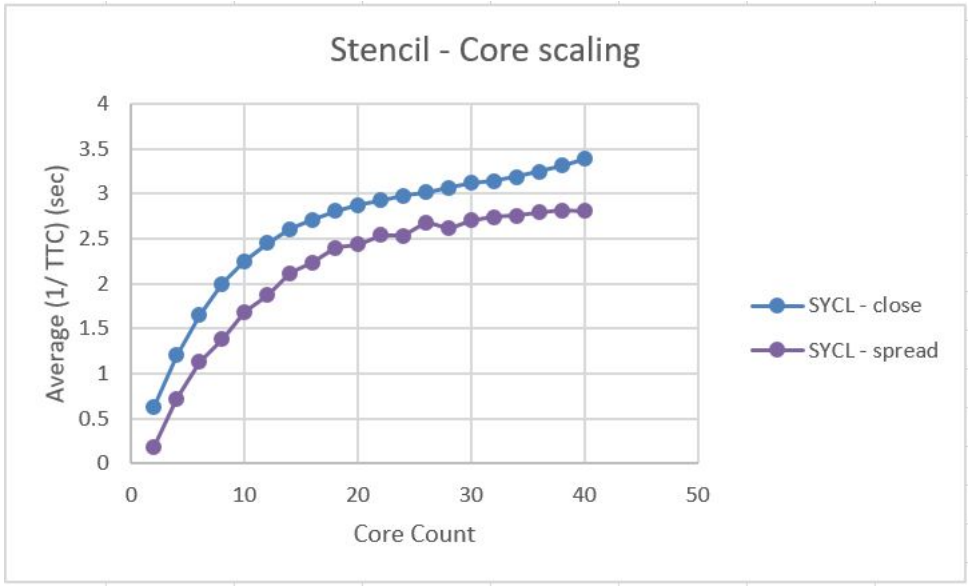

Figure 2: SYCL Core scaling study on 5-pt Stencil.

3.1.3 5-pt Stencil

Discretization is often phrased as applying the difference stencil. Given a physical domain, we apply the stencil to each point in that domain to derive the equation for that point. The 5-pt stencil or 5-pt difference stencil applies the [0,-1,0,-1,4,-1,0,-1,0] vector product to each element in the square domain.

Implementation 1 - The first implementation with size of square domain 40000x40000 comapres two different implementations of Core scaling.
SYCL-close : threads are pinned to CPU cores successively through available cores
SYCL-spread : threads are sread to available cores
Further, we compare the SYCL implementations with the OpenMP implementation of the 5-pt Stencil Kernel.

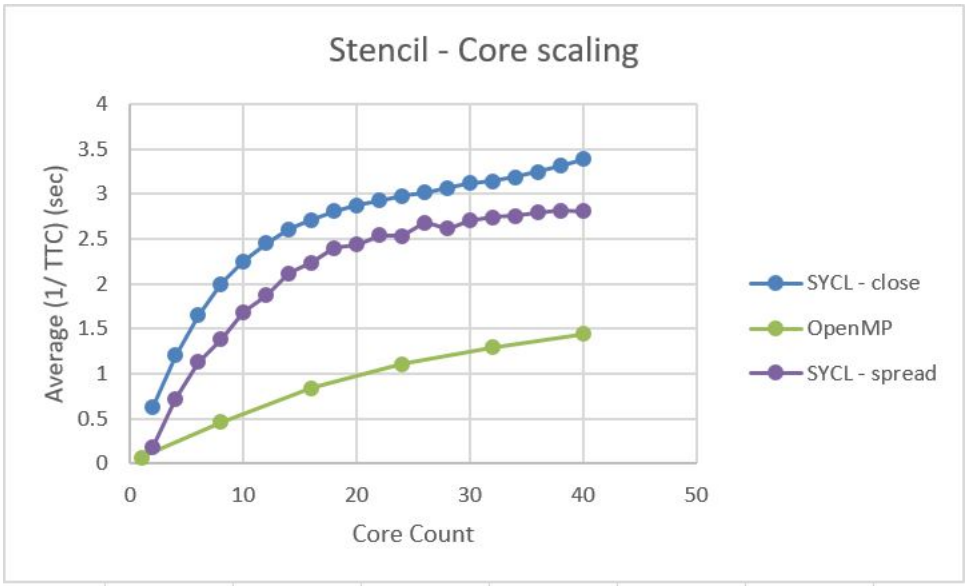

Figure 3: Core scaling study on 5-pt Stencil SYCL vs OpenMP

3.2 GPU

3.2.1 Matrix Multiplication

Implementation 1 - The first implementation with Input Matrices A (8192xK) and B (Kx8192) with varying K from 64 to 4096 displays a monotonically rising graphs.

N	M	K	TTC (Sec)
8192	8192	64	0.05648
8192	8192	128	0.05903
8192	8192	256	0.08414
8192	8192	512	0.13699
8192	8192	1024	0.23906
8192	8192	2048	0.46991
8192	8192	4096	1.87249

Implementation 2 - The second implementation with Input Matrices A (8192xK) and B (Kx8192) with varying K from 64 to 4096 displays a monotonically rising graph.

N	M	K	TTC (Sec)
8192	8192	64	0.04883
8192	8192	128	0.04912
8192	8192	256	0.07327
8192	8192	512	0.12714
8192	8192	1024	0.22595
8192	8192	2048	0.44948
8192	8192	4096	1.84122

Implementation 3 - The third implementation with Input Matrices A (8192xK) and B (Kx8192) with varying K from 64 to 4096 displays a monotonically rising graphs.

N	M	K	TTC (Sec)
8192	8192	64	0.03616
8192	8192	128	0.04409
8192	8192	256	0.07072
8192	8192	512	0.12075
8192	8192	1024	0.21993
8192	8192	2048	0.44721
8192	8192	4096	1.81316

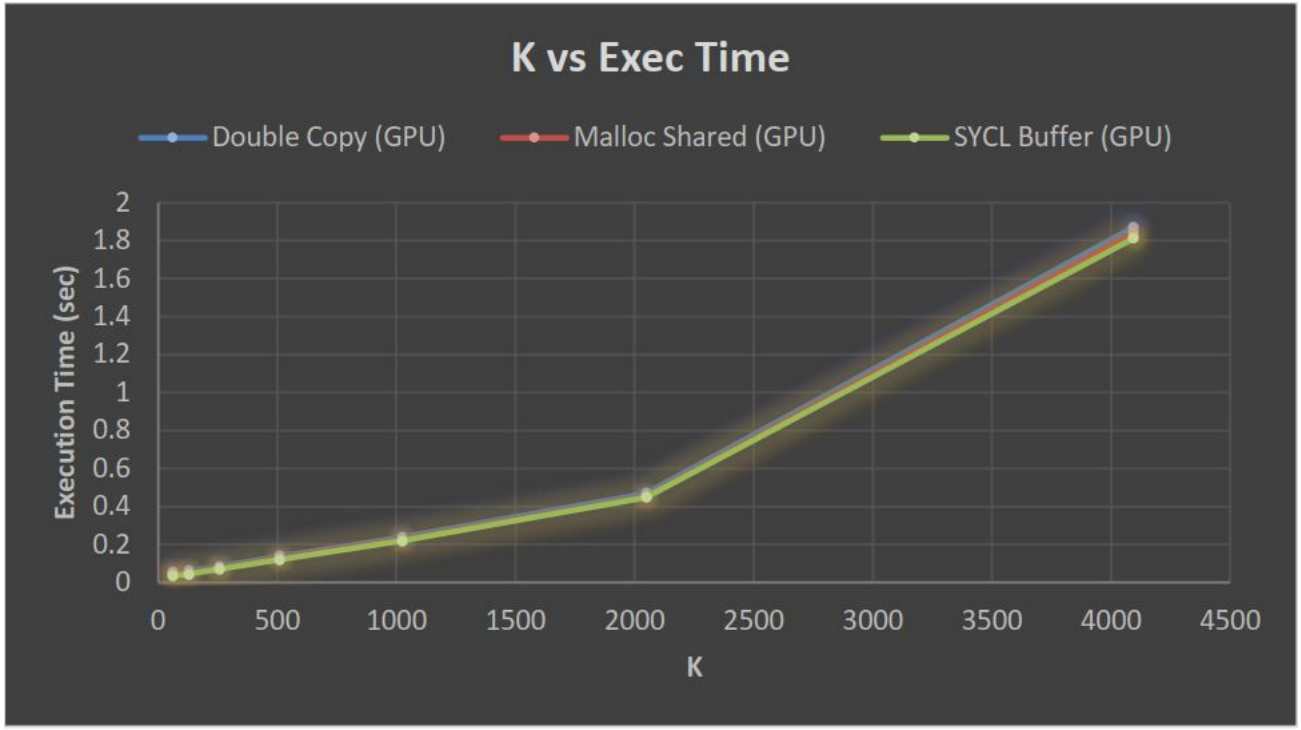

Figure 4: Compute Scaling on a GPU (K in MxNxK) across 3 different memory management techniques

3.2.2 Summary

We notice that for all K, SYCL Buffers perform better than the traditional malloc_* implementations on a GPU.

4 Summary

In this section we summarize and compare the results of the 3 implementations of SYCL Memory Management on a CPU and a GPU.

4.1 Implementation 1

In this implementation, the data movement on a GPU is bottleneck by the PCIe link, while the transfer on the CPU happens relatively quicker on the same node. If the transfer was happening on to another node, it would be bottleneck by the inter-cpu interconnect. Overall, Implementation 1

Figure 5: Compute Scaling on CPU vs GPU for Double Copy

on the CPU is mostly limited by compute, while that on the GPU is limited by data transfer bandwidth.

N	M	K	TTC (Sec)	Mem Transfer (% if TTC)
8192	8192	64	0.05648	1.067
8192	8192	128	0.05903	1.924
8192	8192	256	0.08414	2.380
8192	8192	512	0.13699	3.098
8192	8192	1024	0.23906	3.285
8192	8192	2048	0.46991	3.261
8192	8192	4096	1.87249	1.572

(a) Double Copy - GPU

N	M	K	TTC (Sec)	Mem Transfer (% if TTC)
8192	8192	64	0.4002	0.112
8192	8192	128	0.4288	0.121
8192	8192	256	0.3557	0.200
8192	8192	512	0.3588	0.290
8192	8192	1024	0.3983	0.537
8192	8192	2048	1.4838	0.381
8192	8192	4096	4.1014	0.279

(b) Double Copy - CPU