15-418/618
Final Project
Optimizing Scale Invariant Feature Transform for an Embedded GPU

Prithvi Suresh (Andrew ID psuresh2)
David Wang (Andrew ID dwang3)

Abstract

The aim of the project was to understand and analyse the differences in optimising an application that is meant to run on an embedded GPU. The pervasive use of cheap, affordable GPUs in many real-world applications motivates the need to take a closer look at how algorithms are being run on these devices. Despite the tremendous strides made in increasing the compute capability of these devices (insert reference to survey papers), the increase in complexity of the algorithms (think larger Deep Learning Models) deployed on these devices make it hard to meet real-time constraints. These real-time constraints include real-timeliness, power consumption and reliability. However, by taking a closer look at the features offered by these devices, one can look to optimise applications with these constraints in mind. To this end, we look to optimise an Image processing algorithm on an Nvidia Jetson Nano (insert reference to this). We choose Scale Invariant Feature Transform (insert reference) (SIFT), a keypoint detector widely used in object tracking and detection.

1 Introduction

With the growing affinity for AI/Robotics in almost all applications, there is an increasing need for faster, smaller, and more power efficient edge devices. The capabilities of these edge devices pale in comparison to large gaming GPUs located in server farms. Nvidia has attempted to bridge this gap by releasing a suite of embedded GPUs. These embedded devices are optimised to run machine learning inference at low power. The toolchain to develop applications on these devices is virtually similar to developing on a traditional large discrete Nvidia GPU. Through this project, we aim to explore the effects a resource constrained device can have on parallelizing approaches. To this end we select an embedded device and an image processing algorithm to parallelize.

Image Processing on edge devices proves beneficial for a wide range of applications such as robotics and surveillance. We choose the Scale Invariant Feature Transform (SIFT) [1] algorithm, a widely known key-point detection algorithm to parallelize. SIFT is widely used in image processing to extract key points for object detection, localization, and tracking. SIFT also has huge scope for parallelism and can take advantage of the capabilities of a GPU.

The summary of steps involved in SIFT is as follows

1. **Scale Space Construction**: Construct different scales and different blur levels for a single image
2. **LoG**: Obtain the Laplacian of the Gaussian of each of these images and extract local minima and maxima.
3. **Threshold extrema**: Remove noisy extrema such as edges
4. **Assign Rotation Invariance**: Make extrema invariant to rotation
5. **Assign feature**: Assign a feature vector to the point

The following points describe the contributions we made through the course of this project;

1. We identified pain points in running a highly parallel application on an embedded GPU
2. We took a closer look at the hardware, its capabilities and constraints that it offered, and try to incorporate these into optimising the application
3. We optimise SIFT for video applications, demonstrating the ability of SIFT to run in real-time

2 Method

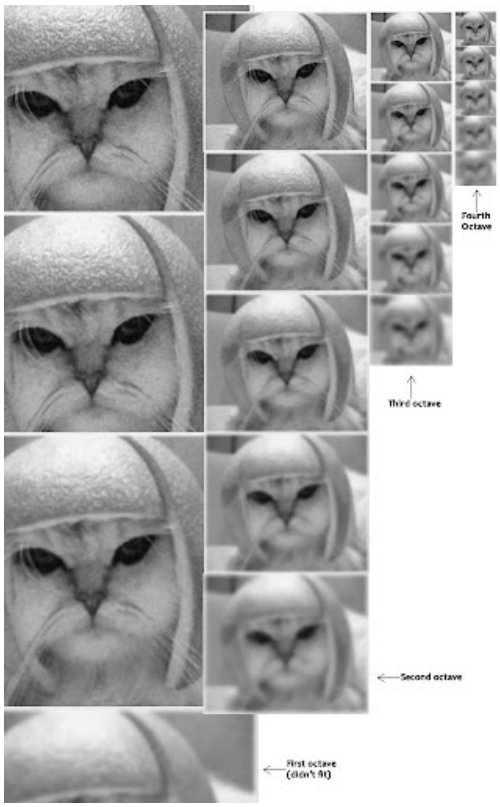
2.1 Scale Invariant Feature transform

Matching different features across different images has always been an issue in computer vision. It is especially difficult when the images given have different scales and orientations. This is where Scale Invariant Feature Transform (SIFT) comes into play. When scale, rotation, illumination, and viewpoints are different across different images, SIFT is still able to find common feature keypoints across different images.

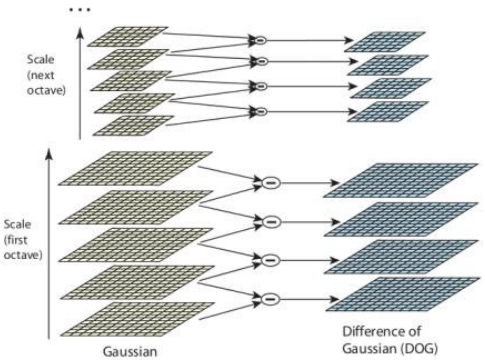
SIFT is an involved algorithm and it can be split into these following parts: constructing a scale space, LoG approximation, finding Keypoints, get rid of bad keypoints, assigning an orientation to keypoints, and generate SIFT Features.

A brief overview of the algorithm is shown below

1. Constructing a scale space: We are essentially preparing the image to ensure the scale invariance by progressively blurring out the images, and then resizing the image by half and blurring out the image again. Figure 1a shows an example of scale space.
2. Then we approximate Laplacian of Gaussian by taking the difference between the scales in each octave of the scale space created earlier. Keypoints are found, which are the maxima and minima in the Difference of Gaussian image. Figure 1b shows the computation of this step.
3. After this, we clean up the key points a bit by eliminating edges and low contrast regions to make the algorithm more efficient and robust. We then have to calculate an orientation for the key points



(a) SIFT scale space



(b) Laplacian of Gaussian (or difference of Gaussian

Figure 1: SIFT

Figure 2: Resultant Keypoints

to make it rotation invariant. Finally, we generate one more representation to help uniquely identify features by creating a fingerprint for each keypoint. Figure 2, shows an example of resultant keypoints from our implementation.

2.2 SIFT on cuda

Initially, we had sought out to implement SIFT from scratch using cuda. However, on reevaluating our aim which was to study the optimisations on embedded GPUs, we identified a pre-existing code base that implemented SIFT to run on images from scratch. The link to the github repository can be found here. The code base implements all of the steps mentioned in the previous section.

2.2.1 Kernel Analysis

The entire image goes through 4 cuda kernel launches in order to extract SIFT feature descriptors. The image is moved to the GPU before kernel launch.

1. Laplace Cuda: This cuda kernel performs both Gaussian filtering and the approximation of the laplacian of the gaussian as specified by the authors. Since the kernel is symmetric about the center, there is no need to store the entire kernel of coefficients. The developers of the repo separate out the convolution by first taking the sum across the vertical and writing it in a shared buffer. Once all threads accumulate their vertical sum in a shared buffer, we can proceed to accumulate the sum across the image in a horizontal fashion to find the final gaussian blurred image. Once this is carried out for each of the scale spaces in a particular octave, the difference between the corresponding pixel between each scale is taken. This gives us the DoG.

2. Finding Extema cuda kernel: Finding the extrema is particularly simple, since we assign a region of pixels to a thread and each thread finds the extremas within its allocated region. There are 26 comparisons made in a reducible fashion to figure out if a point is an extrema or not.

3. Orientation cuda kernel: This kernel involves assigning a rotation to each extrema. This is a low compute kernel since it only has to deal with the number of extremas computed. Each keypoint is assigned it's own block. Once the gradient is calculated for all pixels around the keypoint, it is binned into histograms to detect the largest change in gradient. Lowe et al specified 36 histograms, but to be in sync with cuda we use 32 to maximise coherent execution.

4. Creating descriptors kernel: Each block handles a particular keypoint and extracts descriptors around it. Finally, the keypoints array is moved to the host.

2.3 Jetson Nano

The Jetson Nano is a tiny embedded GPU by nvidia that supports cuda. This allows for versatile development of cuda applications on this device. However, the specs of this embedded GPU pales in contrast to an actual GPU. The Jetson Nano includes the Tegra X1 SOC. The Tegra X1 contains 4 Arm A57 cores that share a

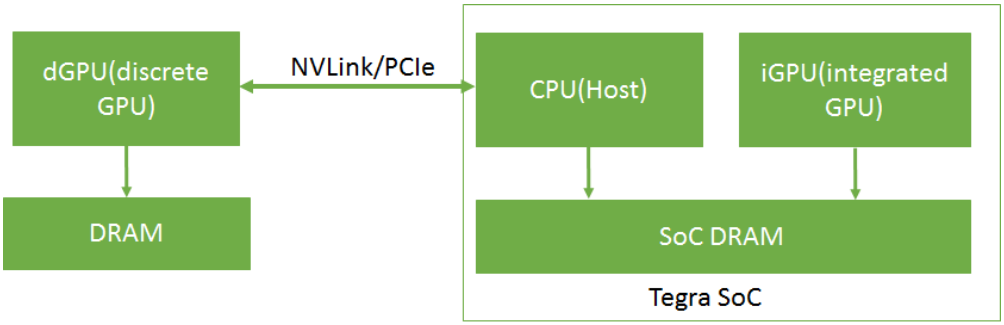


Figure 3: Tegra SOC Memory Layout

2MB cache. It also contains a 128 core integrated Maxwell GPU. Each of these cores are grouped into 32 core blocks each with their own instruction scheduler and so forth. It also has on-chip memory of 4GB[2].

The key difference between the jetson nano and a discrete GPU (dGPU) attached to a powerful CPU is the Tegra X1 memory model (shown in figure 3). A dGPU has its own RAM and any operations to be carried out on a dGPU must have memory transferred to it via PCIe or NVLINK. In contrast, the Memory between the integrated GPU (iGPU) and the CPU is shared. This model has both pros and cons. The cons being the limited RAM that can be accessed by both the GPU and CPU. This means that any memory allocated on the CPU will deduct memory accessed by the GPU as well. The good thing about this design is that there is no need for explicit transfer since all memory is located in the same SOC DRAM.

Just like with any other GPU, the Tegra SOC also consists of different memory regions. Despite all of them being located on the same SOC DRAM, they differ in their caching behaviour. Particularly, Unified Memory is of interest to us since it is accessible by both the CPU and GPU and is also cached on both of them. However, there is a coherency overhead since the jetson Nano lacks I/O coherency. I/O coherency is a feature introduced in devices with compute capability ¿ 7.2 (as defined by Nvidia) where GPUs can read memory from CPU caches. Since the jetson Nano lacks this capability, there is a larger overhead which we will analyze later on in this paper.

2.4 Optimizations

We initially tried optimising SIFT using cuda for images. However, we quickly realised that implementing a naive cuda implementation works just as well on images. The implementation is resource-bound since the Jetson has 128 Maxwell cores. Thus, the given cudaSift implementation runs just as well as it could on a single frame. With the exception of using HostAllocatedPinned memory for SIFT features, we noticed no optimization would lead to a meaningful difference. for a single frame.

However, on running keypoint detections on video streams, we noticed that migrating data between the host and device took up a large portion of the compute(ref next section). Given that the Tegra SOC is capable of cached unified memory, we decided to implement a unified memory buffer for storing frames as they are read and process it in-place on the GPU. Each frame is read from its source into a unified memory pointer. Since the pointer is both accessible by the CPU and the GPU there is no latency of data transfer.

An additional optimization is running asynchronous cuda streams. Kernels running on the default cuda

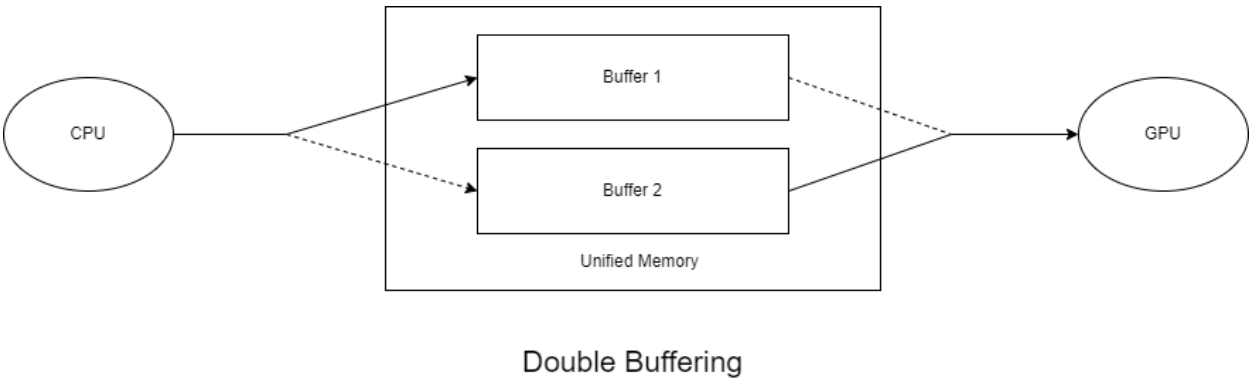


Figure 4: Asynchronous approach

stream synchronises with the host. However, kernels running on different streams do not need to synchronise with the host. This allows us to carry out CPU operations like loading the next frame while the GPU is busy computing on the previous frame. A diagram of this approach is shown in figure 4.

2.5 Dataset

We consider 4 videos at different resolutions. The different resolutions allow us to analyse the performance of the algorithm on different problem sizes. Additionally, each video is of varying complexity, some rich with features and some not so much. This enables us to analyse the effect that the number of keypoints have on the frame rate and the keypoint extraction time.

3 Results

3.1 Metrics

We consider 2 metrics to evaulate our approaches. The *average frame time* (AFT) is defined as the time taken to process an entire frame which includes loading the frame, extracting keypoints and plotting them on the image. This time is indicative as to how we as humans perceive the video. The *average extraction time* (AET) is the time taken to process a frame on the GPU. This time is important since it allows us to quantify the amount of overhead and identify pain points.

3.2 Optimization Approaches

We implement 4 different approaches.

1. Synchronous: Each frame is processed one at a time. Data is moved between the CPU and GPU and back before being displayed.

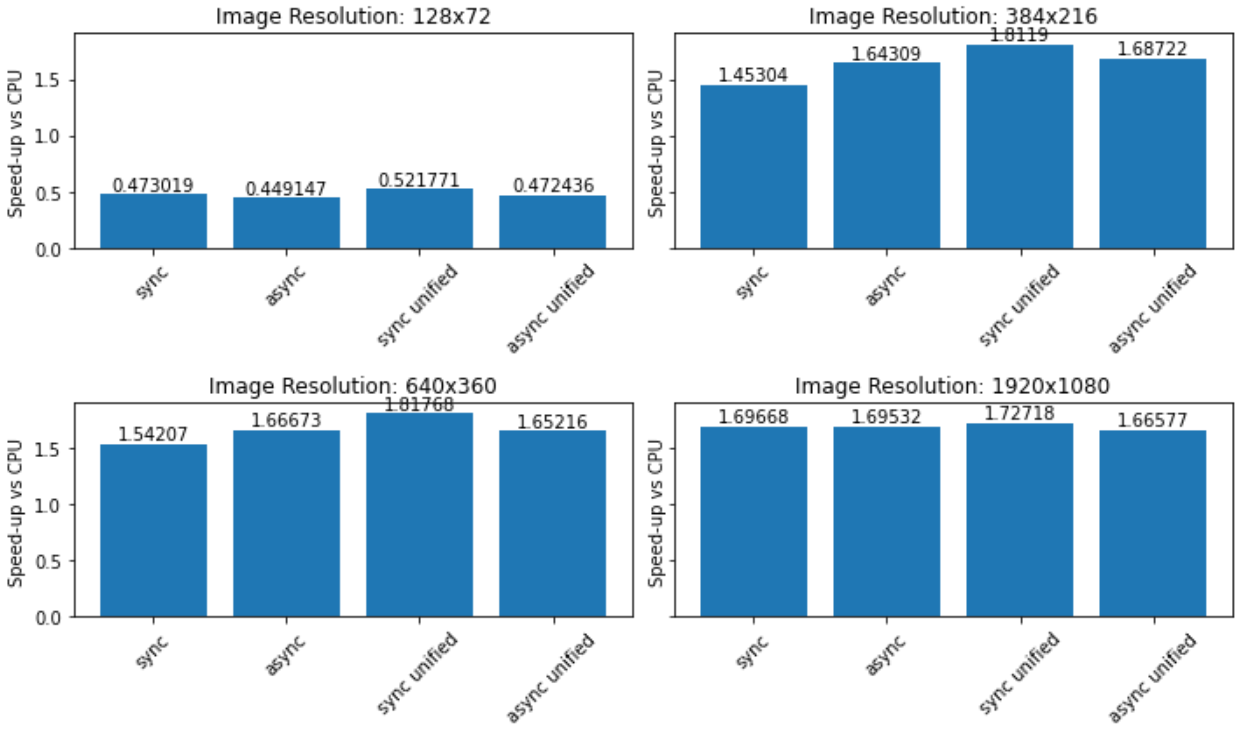


Figure 5: Average Frame time speed-up

2. Asynchronous: This approach uses a double buffer. While the GPU is processing a frame, the next frame is loaded into the CPU before sending it to the GPU.

3. Synchronous Unified: Each frame is processed one at a time. Data is loaded into a unified memory region that is accessible to both the CPU and GPU.

4. Asynchronous Unified: A double buffered approach. Similar to the Asynchronous approach, but 2 unified pointers are used instead. This eliminates the need for explicit movement.

The speed-up for each metric for each of the four approaches is shown. The speed-up is measured against openCV's implementation. The speed-up for both the average frame time and average extraction time is observed.

From figures 6, 5 it is clear that all 4 approaches exceed CPU speed at almost all resolutions. For the smallest resolution, the overhead of launching the kernel outweighs the cost of compute. In this case, the best performance was from the CPU. In all other cases, better speed-up was achieved from the GPU implementation.

Additionally, it is worth noting that as the resolution increases, performance becomes bound by the CPU capability. This can be seen from the figures 7a and**??**. Despite, AET increasing almost linearly, the AFT saturates indicating performance bottlenecks due to the CPU. This is indicative of the fact that performance is bound by the CPU at larger problem complexities.

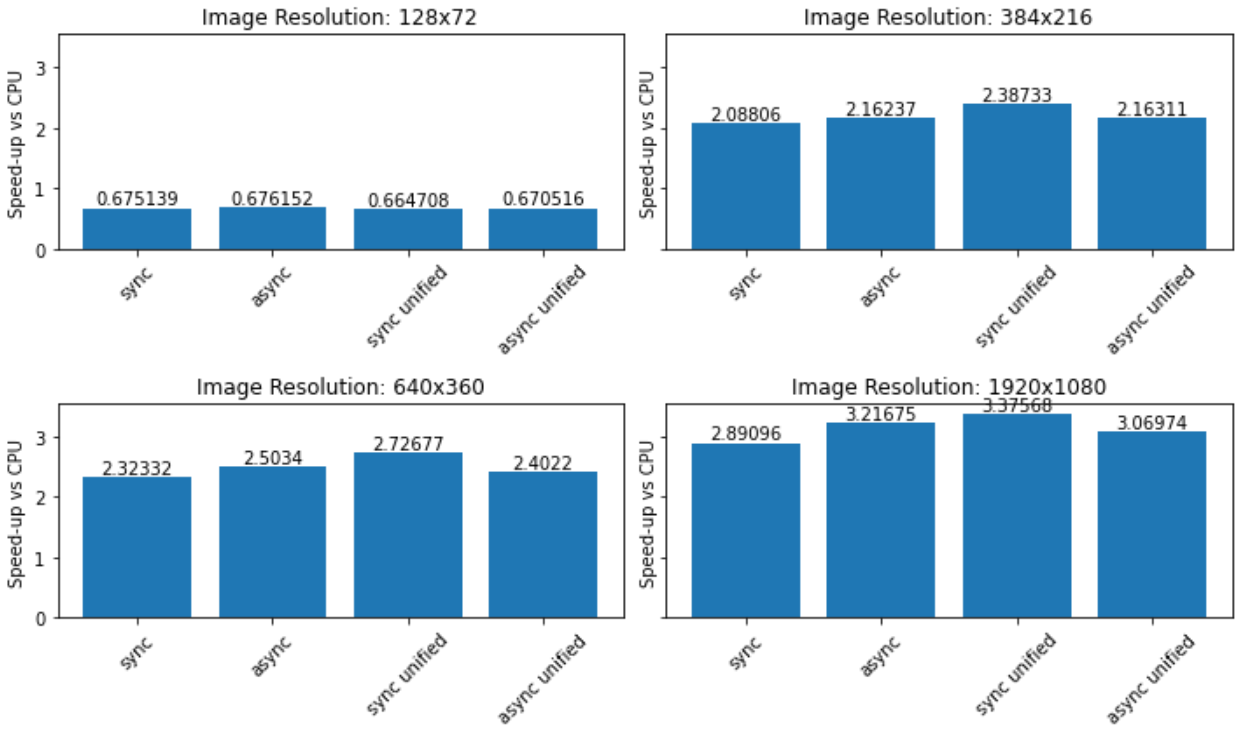
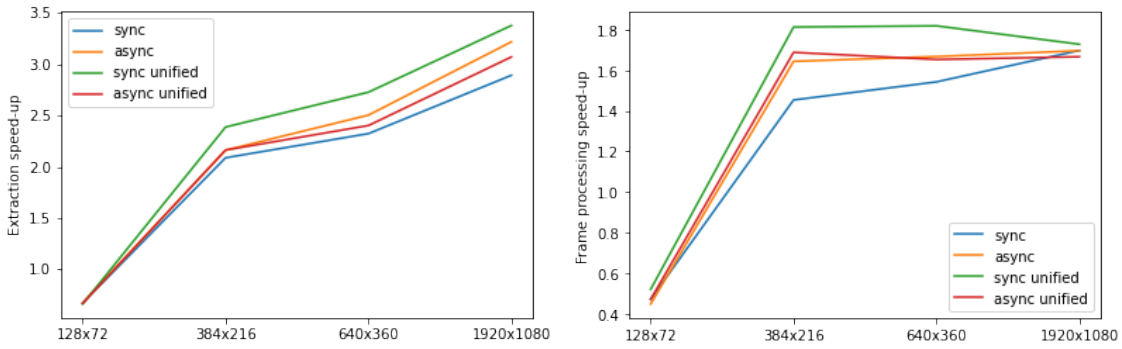


Figure 6: Average Extraction time speed-up



(a) scaling of AET

(b) scaling of AFT

Figure 7: Metric scaling wrt resolution

Type	Time(%)	Time	Calls	Avg	Min	Max	Name
GPU activities:	48.33%	12.3240s	4660	2.6446ms	41.354us	37.870ms	LaplaceMultiMem(float*, float*, int, int, int, int)
	15.17%	3.86870s	2798	1.3827ms	208ns	11.502ms	[CUDA memcpy HtoD]
	13.10%	3.33975s	4660	716.69us	19.114us	6.6094ms	FindPointsMultiNew(float*, SiftPoint*, int, int, int, float,
	10.86%	2.76952s	932	2.9716ms	2.8369ms	11.037ms	LowPassBlock(float*, float*, int, int, int)
	10.58%	2.69773s	3728	723.64us	39.635us	7.4778ms	ScaleDown(float*, float*, int, int, int, int)
	1.33%	339.06ms	4660	72.760us	30.939us	474.43us	ExtractSiftDescriptorsCONST(__int64, SiftPoint*, float, int)
	0.62%	158.67ms	4660	34.050us	21.408us	149.80us	ComputeOrientationsCONST(__int64, SiftPoint*, int)
	0.00%	1.2545ms	932	1.3460us	833ns	2.7090us	[CUDA memset]
	0.00%	1.1378ms	932	1.2200us	989ns	3.4380us	[CUDA memcpy DtoH]
API calls:	73.09%	20.9814s	932	22.512ms	21.331ms	71.345ms	cudaMemcpyAsync

(a) Non-unified profiling results

Type	Time(%)	Time	Calls	Avg	Min	Max	Name
GPU activities:	56.99%	12.3190s	4660	2.6436ms	41.303us	37.948ms	LaplaceMultiMem(float*, float*, int,
	15.28%	3.30303s	4660	708.80us	19.063us	6.6229ms	FindPointsMultiNew(float*, SiftPoint
	12.93%	2.79435s	932	2.9982ms	2.8646ms	14.534ms	LowPassBlock(float*, float*, int, in
	12.48%	2.69788s	3728	723.68us	39.532us	8.5383ms	ScaleDown(float*, float*, int, int,
	1.57%	340.06ms	4660	72.973us	31.511us	479.23us	ExtractSiftDescriptorsCONST(__int64,
	0.73%	158.22ms	4660	33.951us	21.823us	157.40us	ComputeOrientationsCONST(__int64, Si
	0.01%	1.4118ms	1866	756ns	208ns	4.1150us	[CUDA memcpy HtoD]
	0.00%	823.09us	932	883ns	572ns	4.4270us	[CUDA memcpy DtoH]
	0.00%	648.18us	932	695ns	521ns	4.2190us	[CUDA memset]

(b) unified profiling results

Figure 8: Profiling results

3.3 Why does synchronised unified memory do best?

The synchronised unified memory approach performs the best, although we expected the asynchronous unified memory approach to perform the best. On closer inspection, it was found that cuda streams cannot execute concurrently unless GPU resources are free. Since for almost all cases, the 128 core Maxwell GPU that runs 2048 threads at a time is utilised to its fullest, concurrent execution defaults to sequential execution. Additionally, locality takes a hit due to the double buffering.

3.4 Profiling

The results of profiling the GPU activity in all kernels from the approaches is shown in figures

The profiling results from the approach that explicitly moves memory between the CPU and GPU shows that 15% of the time is spent in moving data from the host to the device. This is a large overhead that can be eliminated from using unified memory. Since both GPU and Host memory are located on the same SOC, there is no need for an explicit transfer. Accordingly, the results from the approaches using unified memory similarly also shows that the memcpy overhead is mostly insignificant. Since this memory is additionally cached, it proves particularly useful and hence speeds up our application significantly.

This coincides with the theoretical speed up from reducing 15% of the execution time in all the extraction speed-up figures.

An additional point worth noting is that Jetson Nano does not provide I/O coherency. I/O coherency in Nvidia GPUs is the ability of the GPU to access the CPU cache. Using unified memory might work best in a jetson Xavier, which is capable of I/O coherency. The overhead to access unified memory will be lower, and we can see significant speed gains in the double buffered approach.

4 Conclusion

The aim of the project was to understand the approach to optimise a particular application to run on an embedded GPU. Through the course of this project we have made the following conclusions.

1. Embedded GPUs are smaller and thus cannot fit separate SOCs for gpu and cpu memory. This allows developers to take advantage of the needlessness of memory movements. However, developers need to be aware of the synchronization mechanisms provided in the board.
2. Memory is precious. The limited RAM coupled with expensive memory calls requires diligent use of memory sys calls. This preallocating buffers before proves incredibly useful.
3. Performance can quickly be bounded by CPU performance. This coupled with the fact that an embedded GPU is smaller than normal, problems might not scale as well as one would expect it to.
4. Kernel Launches can be overlapped with CPU operations. This is especially useful for real-time applications. However, concurrent execution is only possible when there are GPU resources available. A large problem might not experience a large gain from concurrent CPU and GPU execution.

5 Acknowledgements

Thanks to Marten Bjorkman for providing an initial implementation. Additionally, we would like thank Dipan Pal for providing us with the hardware necessary for the project's completion.

References

[1] G Lowe. Sift-the scale invariant feature transform. *Int. J*, 2(91-110):2, 2004.

[2] Nvidia. Jetson nano datasheet.