CSE237C Project 4 - Matrix Multiplication on Intel DevCloud Using DPC++

Henrik Larsson Hestnes

November 2021

1 Matrix Multiplication on Intel DevCloud Using DPC++ - Introduction

The purpose of this project was to get an introduction to hardware acceleration using Intel DevCloud. Matrix multiplication is a canonical example in parallel computing, and we will therefore use this to explore the opportunities Intels's oneAPI, or more specific DPC++(Data Parallel C++), gives us to perform hardware acceleration. All the throughputs in this report is calculated using the "End Cycle" of the calculation kernel and "Compile Estimated Frequency" from the generated report as follows:

$Throughput[Hz] = \frac{Compile\,Estimated\,Frequency[Hz]}{End\,Cycle(latency)[\#cycles]}$

1.1 Q&A

Question 1: Load-Store Unit: What are the different options for the Load-Store Units? Describe how the LSU style affects the c_calc latency. Which LSU style provides the lowest latency? Why?

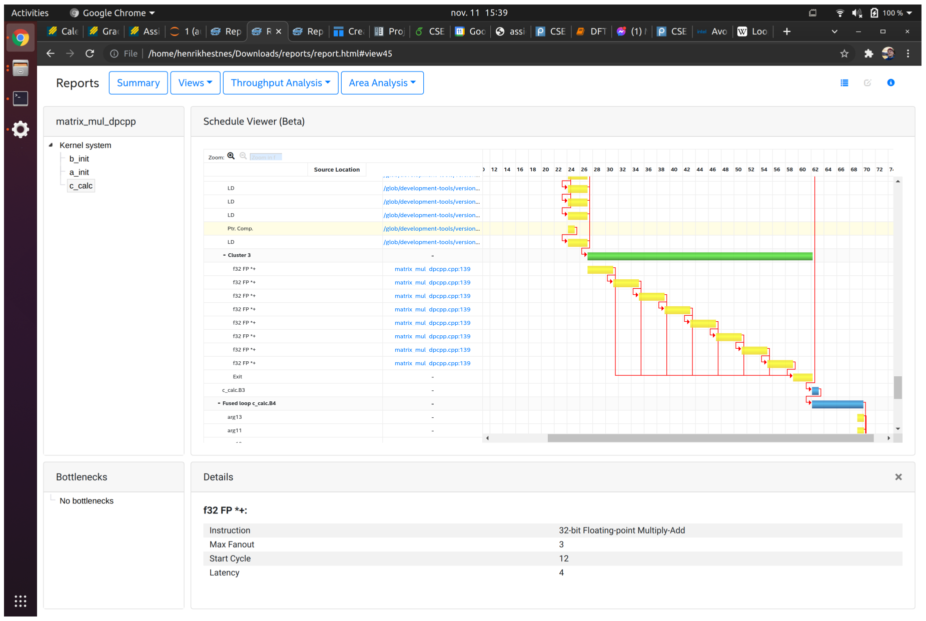
LSU style	**Throughput**	**ALUT**	**REG**	**MLAB**	**RAM**	**DSP**
Burst-Coalesced	902 kHz	16214	31729	175	105	30
Prefetching	5.33 MHz	15926	28494	199	89	30
Pipelined	4.80 MHz	15342	27798	203	63	30

There are three different option with regards to the Load-Store Unit style. The default Burst-Coalesced style clearly performs worst in this case, with a way lower throughput and slightly higher resource usage. The two other LSU styles, prefetched and pipelined, performs more similar. Both significantly reduces the latency of c_calc, and thus improves the throughput of the matrix-multiply. The LSU style which provides the lowest latency is the prefetched, with only 45 cycles of latency at a clock frequency of 240 MHz. The reason that the Burst-Coalesced style performs so much worse than the two others is that it waits until the largest possible memory burst can be made before making the transaction. The larger the burst, the more requests has to be aquired before the transaction can be made. The latency is thus determined by the size of this burst, which is not efficient in this case. Since the Prefetced Style performs so much better in almost every aspect of this matrix multiplication, this will be used as baseline for the coming optimizations. This optimization can be seen in the folder mm_optimized1.

Question 2: What are the effects and general trends of performing unrolling using the pragma? Are the results as expected?

Unroll factor	Throughput	ALUT	REG	MLAB	RAM	DSP
0	5.33 MHz	15926	28494	199	89	30
2	4.90 MHz	17014	30476	217	115	32.5
4	4.21 MHz	19215	34252	251	167	37.5
8	3.12 MHz	24011	43403	268	290	47.5

For this task, the m_size was changed to 256, which gave matrix A dimension 32x64 and matrix B dimension 64x 128, and thus C will get dimensions 32x128. The following ”optimization” with unrolling factor 8 can be seen in mm_optimized2.



Details

f32 FP *+:

Instruction	32-bit Floating-point Multiply-Add
Max Fanout	3
Start Cycle	12
Latency	4

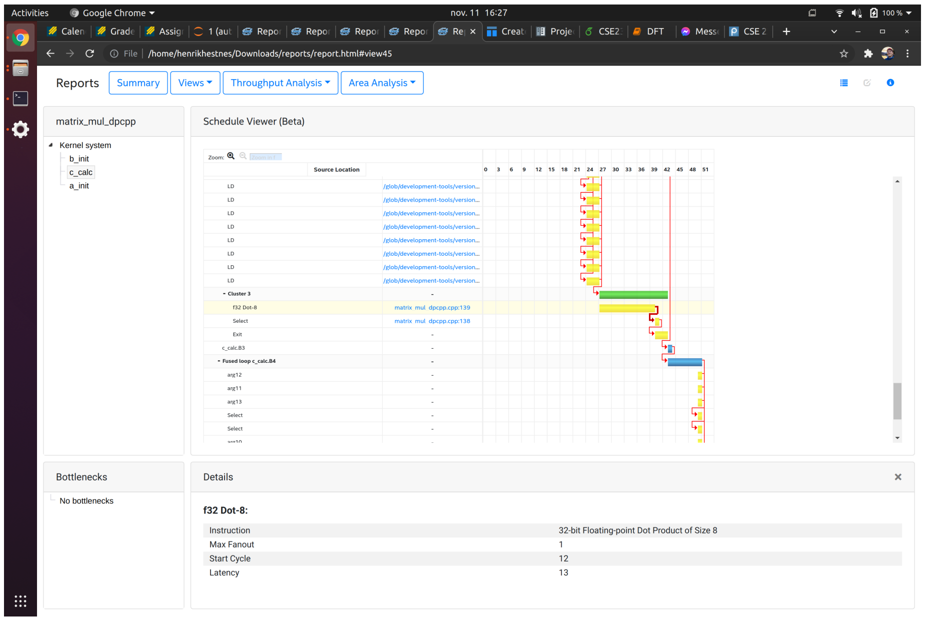
Figure 1: Schedule viewer of mm_optimized2

The general trends and effects of performing unrolling using the pragma is as follows; the higher unroll factor, the higher throughput and the higher resource usage. Looking at the table above, it is clear that this is not the case this time. The resource usage goes as expected up. However, the latency of c_calc goes up with higher unrolling factor, and therefore the throughput goes down. This is not what I expected with regards to the throughput, as I expected it to go up. The reason for the decrease in throughput can be seen in the schedule viewer in the figure above. There you can see that the 32-bit Floating-point Multiply-Add is implemented sequentially, thus becoming a bottleneck, while in the not unrolled version it manages to do the MAC's in parallel. I expected and hoped that the multiply-accumulate in the unrolled loop would happen in parallel as a reduction three, but the tool did not implement them as that, which lead to a lower throughput than the sequential version. However, even if these were implemented totally in parallell, this would not alone lead to a increase in throughput, so to improve the throughput one have to find other improvements. I do however think the tools implementation of the baseline is quite good, and the computation of the MAC without unrolling do lead to a relatively low lateny.

Question 3: What are the effects and general trends of performing manual unrolling? Are the results as expected?

Unroll factor	Throughput	ALUT	REG	MLAB	RAM	DSP
0	5.33 MHz	15926	28494	199	89	30
2	5.00 MHz	16913	30172	218	115	32.5
4	4.71 MHz	18871	33433	247	167	37.5
8	4.44 MHz	22884	40653	306	271	47.5

For this task, the m_size was changed to 256, which gave matrix A dimension 32x64 and matrix B dimension 64x128, and thus C will get dimensions 32x128. The following ”optimization” with manual unrolling factor 8 can be seen in mm_optimized3.



Details

f32 Dot-8:

Instruction	32-bit Floating-point Dot Product of Size 8
Max Fanout	1
Start Cycle	12
Latency	13

Figure 2: Schedule viewer of mm_optimized3

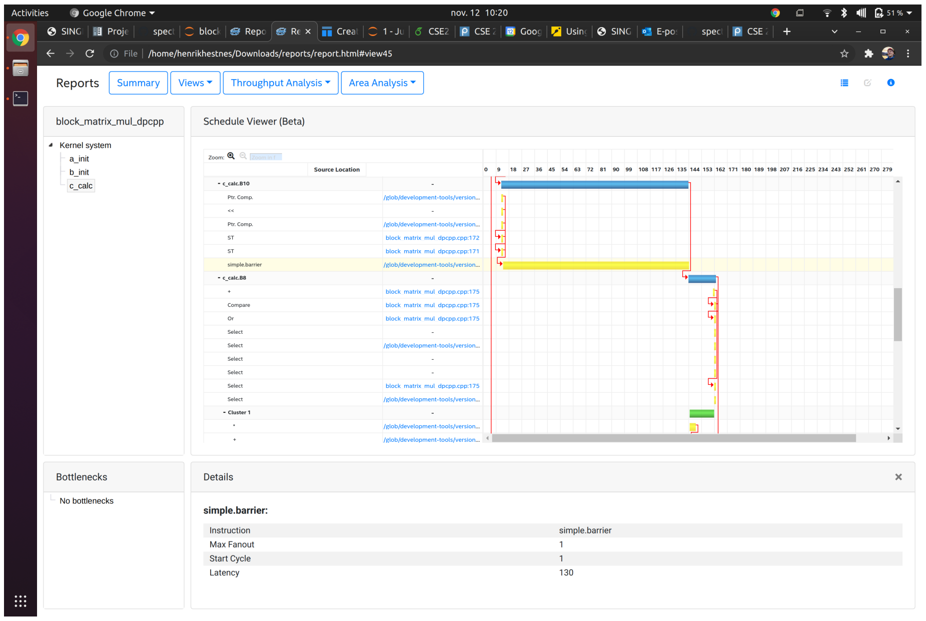
Also with the manual unrolling, both the latency and resource usage goes up. However, the increase in latency is not that significant as for the pragma unroll. The reason for this can also be seen in the schedule viewer above, where the MAC is implemented as a 32-bit Floating-point Dot Product of Size 8 with a latency of 13, instead of 8 times a latency of 4 as with the pragma unroll. This is a very banal example of that the compiler is not necessarily smarter than a human. The decrease in MAC latency obviously leads to a lower latency compared to the pragma unroll, but it still does not beat the baseline, which again tells me that the baseline implementation is quite good.

2 BMM: Block Matrix Multiplication(Question 4?)

As I think the project description is ambiguous to whether the matrix size should be a knob in the Design Space Exploration from the start or just for the final, best design, I will only explore the effect of the matrix size on the final design.

The first optimization made in this design was to change the LSU from Burst-Coalesced to Prefetched, which this time lead to a significant 147% increase in throughput. I also tried different LSU styles on the coming design space exploration, but the Prefetched always came out on top.

In this task we also have a bunch of other optimization choices. By trying to tweak the given knobs, it does not take long time to see that increasing the unroll factor also here leads to an increase in latency.



	Source Location
c_calc.B10	-
Ptr. Comp.	/glob/development-tools/version...
<<	-
Ptr. Comp.	/glob/development-tools/version...
ST	block_matrix_mul_dpcpp.cpp:172
ST	block_matrix_mul_dpcpp.cpp:171
simple.barrier	/glob/development-tools/version...
c_calc.B8	-
+	block_matrix_mul_dpcpp.cpp:175
Compare	block_matrix_mul_dpcpp.cpp:175
Or	block_matrix_mul_dpcpp.cpp:175
Select	-
Select	/glob/development-tools/version...
Select	-
Select	-
Select	block_matrix_mul_dpcpp.cpp:175
Select	/glob/development-tools/version...
Cluster 1	-
*	/glob/development-tools/version...
+	/glob/development-tools/version...

simple.barrier:

Instruction	simple.barrier
Max Fanout	1
Start Cycle	1
Latency	130

Figure 3: Schedule viewer of mm_optimized2

The biggest problem for this implementation can be seen in the schedule viewer above. For every iteration of the outer for-loop, every work-item within the work-group has to wait for the other work-items at the barrier both before and after the inner for-loop, such that they are synchronized. Both of these barriers has an excessive latency of 130 cycles, which is undoubtedly the bottleneck of this baseline. One thing that can be done to improve the throughput is to remove the second barrier, which is the least important. I tested it out, with success both in terms of improved throughput and result of the BMM. By doing that you do however run the risk that one or several work-items might overwrite the local memory before another work-item have read it, which might again lead to a wrong result of the BMM. Since I find a BMM which sometimes computes wrong matrices quite useless, this was not an optimization I went further on. I do however believe that having to wait a total of 260 cycles at the two barriers is way too much, and definitely something that should be improved.

As in the previous MM tasks, the compiler this time also infers the MAC's of the manual unrolling of factor 8 as an efficient "32-bit Floating-point Dot Product of Size 8". One interesting difference between the BMM and the MM is however how the pragma loop unrolling is handled. For some reason the compiler understands that the efficient "32-bit Floating-point Dot Product of Size 8" is better also for the pragma this time, which is the reason why the throughput does not change whether the unroll factor is of type manual or pragma.

Another interesting thing is, as you can see from the table below, that the difference between the block sizes of 16 and 32 is nothing in terms of throughput and latency, and almost negligible in terms of resource usage, except for the design that combines the manual and pragma unroll. Even the block sizes of size 1 and 128 only lead to a difference of 4 cycles in terms of latency. I would suppose that smaller block sizes would make it easier to take advantage of the opportunities of parallelism, which would lead to a significant increase in both throughput and resource usage. For some reason this does not happen, and I can seem to find the reason why in the report.

Block size	**Pragma unroll**	**Manual unroll**	**Throughput**	**ALUT**	**REG**	**MLAB**	**RAM**	**DSP**
16	0	0	792 kHz	25089	46986.9	293	279	37
16	2	0	782 kHz	25910	48252.9	319	285	48
16	0	2	782 kHz	25910	48252.9	319	285	48
16	8	2	759 kHz	47433	91467.9	197	444	202
32	0	0	792 kHz	25087	46985.9	293	279	37
32	2	0	782 kHz	25908	48251.9	319	285	48
32	0	2	782 kHz	25908	48251.9	319	285	48
32	8	2	759 kHz	38444	66624.9	633	445	202

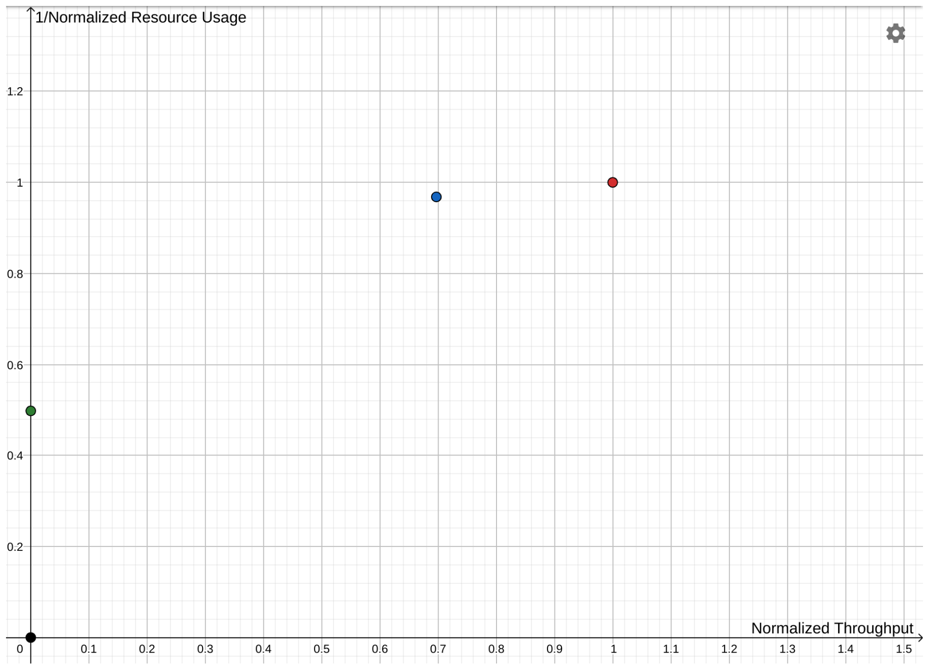


Figure 4: Normalized chart of the above architectures

Above you can see the results of the design space exploration. Since the result of tweaking on the unroll factor became as it became, there is unfortunately no doubt which of these results are the best. The normalized chart also becomes somewhat superfluous because of the results, but I made it anyway. The main reason of making the normalized chart is to make it easier to see which points are at the Pareto frontier, but we do not get the clear Pareto frontier one usually gets by doing the design space exploration due to the results. The two red points have lower resource usage and hgher throughput than any of the other, which make them superior. The points in the chart are coded with the same colors as the table above it to make it easy to tell which is which, if there even were any doubts. The red ones are as said previous obviously the best, and I will therefore try out different matrix sizes with these two, to see whether the block size has a bigger impact on larger matrices.

Block size	Matrix size	Throughput	ALUT	REG	MLAB	RAM	DSP
16	128	792 kHz	25089	46986.9	293	279	37
16	256	792 kHz	25087	46985.9	293	279	37
16	512	792 kHz	25086	46984.9	293	279	37
16	1024	792 kHz	25084	46983.9	293	279	37
16	2048	792 kHz	25083	46982.9	293	279	37
16	4096	792 kHz	25081	46981.9	293	279	37
128	4096	792 kHz	25077	46978.9	293	279	37

As can be seen from the table above, the matrix size does not have any impact in the throughput. The reason for this is that the block size does not change, so it will just happen more calculations in parallel, and thus it will not impact the latency or throughput. What I find more weird is that the resource usage is more or less constant during the changes in matrix size. A doubling in matrix size should lead to a 4 times as big amount of work groups, which would lead to significantly more local matrices and thus an increase in resource usage, but according to the report this does not happen. I also find the fact that the report says that an increase in block size from 16 to 128 on a 4096x4096 does not impact the throughput interesting, as I intuitively would have thought that such an increase in block size would lead to an increase in latency due to more computations per parallel, and thus a decrease in throughput.