02258 Parallel Computer Systems Fall 2022

Third Report

Tomás Estácio
Department of Applied
Mathematics and Computer
Science
Technical University of Denmark
Lyngby, Copenhagen, Denmark
s223187@dtu.dk

1 Databar Exercise 8: Performance Models

For this Databar, I was asked to explore the code balance and machine balance using the OpenCL for the previous Databar reported, using the CPU OpenCL target implementations.

Firstly, I used the kernel code from Exercise01, Exercise02 and Exercise05 and calculated the theoretical value of the code balance by using the formula:

$$Code\ balance\ (Bc) = \frac{data\ traffic\ (Words)}{floating{-}point\ operations\ (Flops)}$$

For this formula, I am required to know the data traffic, which refers to all words transferred over the data path, making it dependable of the hardware, and the number of floating-point operations in the kernel code of each Exercise, that are any mathematical operation or assignment that involves floating-point numbers (numbers with decimals points in them).

In Exercise01 there isn't a kernel code to analyze because the program does not perform arithmetic operations, only prints out information about the device used for the execution of the program, so I didn't calculate the code balance for that exercise.

In Exercise02, there is the following kernel code:

```
const char *KernelSource = "\n" \
"__kernel void vadd(                                    \n" \
"    __global float* a,                                 \n" \
"    __global float* b,                                 \n" \
"    __global float* c,                                 \n" \
"    const unsigned int count)                          \n" \
"{                                                      \n" \
"    int i = get_global_id(0);                          \n" \
"    if(i < count)                                      \n" \
"        c[i] = a[i] + b[i];                            \n" \
"}                                                      \n" \
"\n";
```

Figure 1: Kernel code of program *vadd_c.c* in Exercise02

In each iteration of this kernel code, there are two loads, one for *a[i]* and other for *b[i]*, one store for *c[i]* and one floating-point operation, the addition of *a[i]* with *b[i]*. For those reasons, it is possible to calculate the code balance:

$$Code\ balance\ (Bc) = \frac{2+1}{1} = 3$$

In Exercise05, there is the following kernel code:

```
const char *KernelSource = "\n" \
"__kernel void vadd(                                    \n" \
"    __global float* a,                                 \n" \
"    __global float* b,                                 \n" \
"        __global float* d,                             \n" \
"                    \n" \
"    __global float* c,                                 \n" \
"    const unsigned int count)                          \n" \
"{                                                      \n" \
"    int i = get_global_id(0);                          \n" \
"    if(i < count)                                      \n" \
"        c[i] = a[i] + b[i] + d[i];                     \n" \
"}                                                      \n" \
"\n";
```

Figure 2: Kernel code of program *vadd_c.c* in Exercise05

In each iteration of this kernel code, there are three loads and one store to provide the required input data. There are also two floating-point operations, so the code balance will be:

$$Code\ balance\ (Bc) = \frac{3+1}{2} = 2$$

For the execution of these programs, I used the device Intel(R) Xeon(R) Gold 6126 CPU @ 2.60GHz, which is a CPU from GenuineIntel with a maximum of 24 "compute units". To calculate the machine balance for this CPU, I first need to know two values: the peak memory bandwidth, in GWords/sec, and the peak floating-point performance, in GFlops/sec.

For a realistic calculation of the memory bandwidth of the CPU used, we used the STREAM benchmark program, assigning an array size of 9856000 elements, due to L3 cache's size of 19712000 bytes. The results obtained were the following: for the copy operation, I got a maximum memory bandwidth of 13.9 GB/s; for the scale operation, I got 13.775 GB/s; for the add operation, I got 15.036 GB/s; for the triad operation, I got 14.793 GB/s. From these results, the scale and the add operations are most relevant, because the first has the same code balance as Exercise02 and the second has the same code balance as Exercise05.

Then, to get the value of the peak floating-point performance, I found the value in an Intel website with peak performance in GFLOPS for different processors they have, including the one we are using, getting a value of 652.8 GFLOPS

So, for the CPU executing Exercise02, the machine balance obtained is 0.0211, and for the same CPU executing Exercise05, the machine balance is 0.0230.

Relating now the value of the code balances with the machine balance, I can conclude that the kernel in Exercise02 only reaches 0.703% of the peak performance, and the kernel from Exercise05 only reaches 1.152% of peak performance.

There are some assumptions associated to the performance model selected for this exercise that can explain the low values obtained, like the access latency is assumed to be hidden completely and not using parallelism in the memory operations, which would be a great way to improve it.

2 Databar Exercise 9: Parallelism in Machine Learning

In this exercise, I have input images of 300x300 pixels, with 3 color channels (red, green and blue channels), 3x3 filter kernels, with also 3 channels and a single output channel. With these conditions, to perform a convolution in 3D with no padding and stride of 1, it is just like a convolution in 2D, except it performs the 2D work 3 times, since we have 3 input channels. Using the 300x300 input image and the 3x3 filter, I will have 298*298*9 multiplications for each input channel in the convolutions, being 298*298 the number of convolutions performed and 9 the number of multiplications done in each convolution, so the total number of multiplications is 298*298*9*3 = 2397708 multiplications. In terms of additions, I will have 298*298*8*3+298*298*2 = 2308904 total additions, since we have 298*298 convolutions, 8 additions per convolution, 2 additions per to sum the convolutions in each input channel and 3 input channels considered.

These previous calculations were considering only one output channel, but if I want to perform the same exercise but for 64 output channels, I will have to use 64 more kernel filters of 3x3 with 3 channels, because each filter connects to every input channel and, as I add more filters, it increases the depth of the output image, so the number of operations for both multiplications and additions will be multiplied by 64, getting a total number of 153453312 multiplications and 147769856 additions.

Now, the challenge is to add a fully connected layer between the filter and the output layer in this model, where I can obtain 20 outputs with just one image as input. Therefore, the input image of 300x300 pixels with 3 channels and the filter with 3x3 pixels and 3 channels perform the already described convolution, giving an output of one 300x300 pixels with one channel. As the output obtained has only one channel, I can now take that result and use it as an array (1x600) in the Fully Connected Layer, to calculate the final 20 outputs, using a weights matrix with size of 600x20, getting an output vector of 1x20. For this process, the total number of multiplications will be 2397708+(600*20) = 2409708 and the total number of additions will be 2308904+(600*19) = 2320304.

For this machine learning model, in which there is a considerable amount of both multiplications and additions, the idea of performing these processes using parallelism is a great one, because these operations can be easily parallelized. To achieve the maximum speedup possible, there are some factors to consider, like the temporal architectures , such as CPUs and GPUs, that can reduce the number of multiplications to increase the throughput, which is the most beneficial optimization for our model. CPUs and GPUs use techniques like SIMD (Single Instruction Multiple Data) or SIMT (Single Instruction Multiple Threads) to perform the multiplications and additions in parallel, mapping the Fully Connected Layer and the Convolution Layer to a matrix multiplication and using approaches like the Fast Fourier Transform (FFT), with optimizations like precomputing and storing the FFT of the filter and of the input feature map, computing it once and used to generate multiple output channels, and the Strassen and Winograd approaches, rearranging the computation to reduce the number of multiplications, being possible to get a speedup approximately 2, using the appropriate optimization algorithm.

For this machine learning model, the hardware design space is very important so that we are able to reduce the number of times it is needed to access memory, so that we can save time reusing data that we know we have to access a lot of times, like the input feature map, the convolutional and the filter. So, my proposal would be to save those values in small memory spaces that have smaller costs and are closer to the processor, so that there isn't much time lost in dataflows. The resulting cost for the memory space would be, if in a large memory fits all the temporaries values and weights, 200, plus the small memories that would have to fit 300*300*3 values for the input image, 300*300 for the output of the convolutional and 3*3 for the filter, costing 4219, plus the total number of multiplications and additions of 4730012, assuming no reduction in the number of multiplications, getting a total cost of 4734431, in the worst case scenario. From this result, I assume the speedup would be from the reduction in the data flow, by modelling the memory architecture closer to the processor that is executing the operations, and from the reuse of data that is in smaller memories so that it spends less time looking for the value.

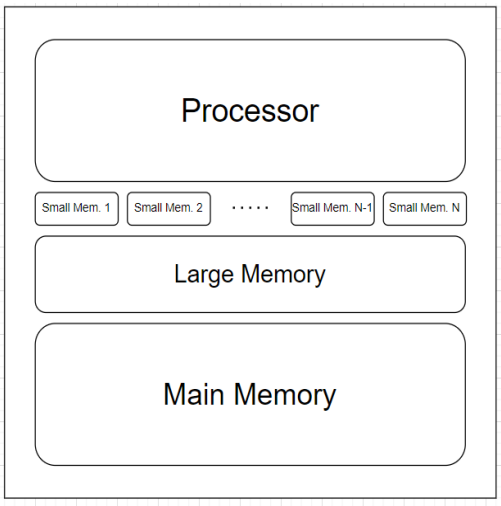

Figure 3: Diagram of memory architecture for machine learning model

REFERENCES

[1] Chapman & Hall, (2019), Introduction to High Performance Computing for Scientists and Engineers.

[2] Intel, Intel Xeon Gold 6126 Processor 19.25M Cache 2.60 GHz Product Specifications, https://ark.intel.com/content/www/us/en/ark/products/120483/intel-xeon-gold-6126-processor-19-25m-cache-2-60-ghz.html.

[3] MicroWay, (09/03/2022), Performance Characteristics of Common Transports and Buses, https://www.microway.com/knowledge-center-articles/performance-characteristics-of-common-transports-buses/.

[4] Red Hat Research, (21/09/2022), Kernel Techniques to Optimize Memory Bandwidth with Predictable Latency, https://research.redhat.com/blog/research_project/kernel-memory-and-latency/.

[5] Panagiotis Antoniadis, (06/11/2022), Baeldung on Computer Science, https://www.baeldung.com/cs/convolutional-layer-size.

[6] Suhyun Kim, (15/02/2019), A Beginner's Guide to Convolutional Neural Networks (CNNs), https://towardsdatascience.com/a-beginners-guide-to-convolutional-neural-networks-cnns-14649dbddce8.

[7] John D. McCalpin, STREAM: Sustainable Memory Bandwidth in High Performance Computers, https://www.cs.virginia.edu/stream/.

[8] Diego Unzueta, (18/10/2022), Fully Connected Layer vs. Convolutional Layer: Explained, https://builtin.com/machine-learning/fully-connected-layer.

[9] IndianTechWarrior, (05/06/2022), Fully Connected Layers in Convolutional Neural Networks, https://indiantechwarrior.com/fully-connected-layers-in-convolutional-neural-networks/.

[10] Vivienne Sze, Yu-Hsin Chen, Tien-Ju Yang, Joel Emer, (13/08/2017), Efficient Processing of Deep Neural Networks: A Tutorial and Survey, https://arxiv.org/pdf/1703.09039.pdf.

[11] Intel, APP Metrics for Intel® Microprocessors, https://www.intel.com/content/www/us/en/support.html.