Guidelines and Conventions

Mikael Hedberg

June 23, 2015

1 Layer Memory Layout

When designing the architecture of the CNN, one of the goals was being able to support extremely large networks with as good performance as possible. For example, when back propagating a convolutional layer, you need either to add zero padding around the unit, or a bunch of conditional statements in the code that emulates zero padding. The latter is hard to maintain in the long run. For GPUs, it's always an bad idea to rely heavily on branching since the instruction set is executed simultaneously over a warp (or wavefront). Furthermore, allowing for padding and memory offsets, we have greater flexibility to align the memory for either 1) optimized memory access for GPUs or 2) vectorization for CPUs. The decision was taken to impose the below memory model that must be handled by the layers.

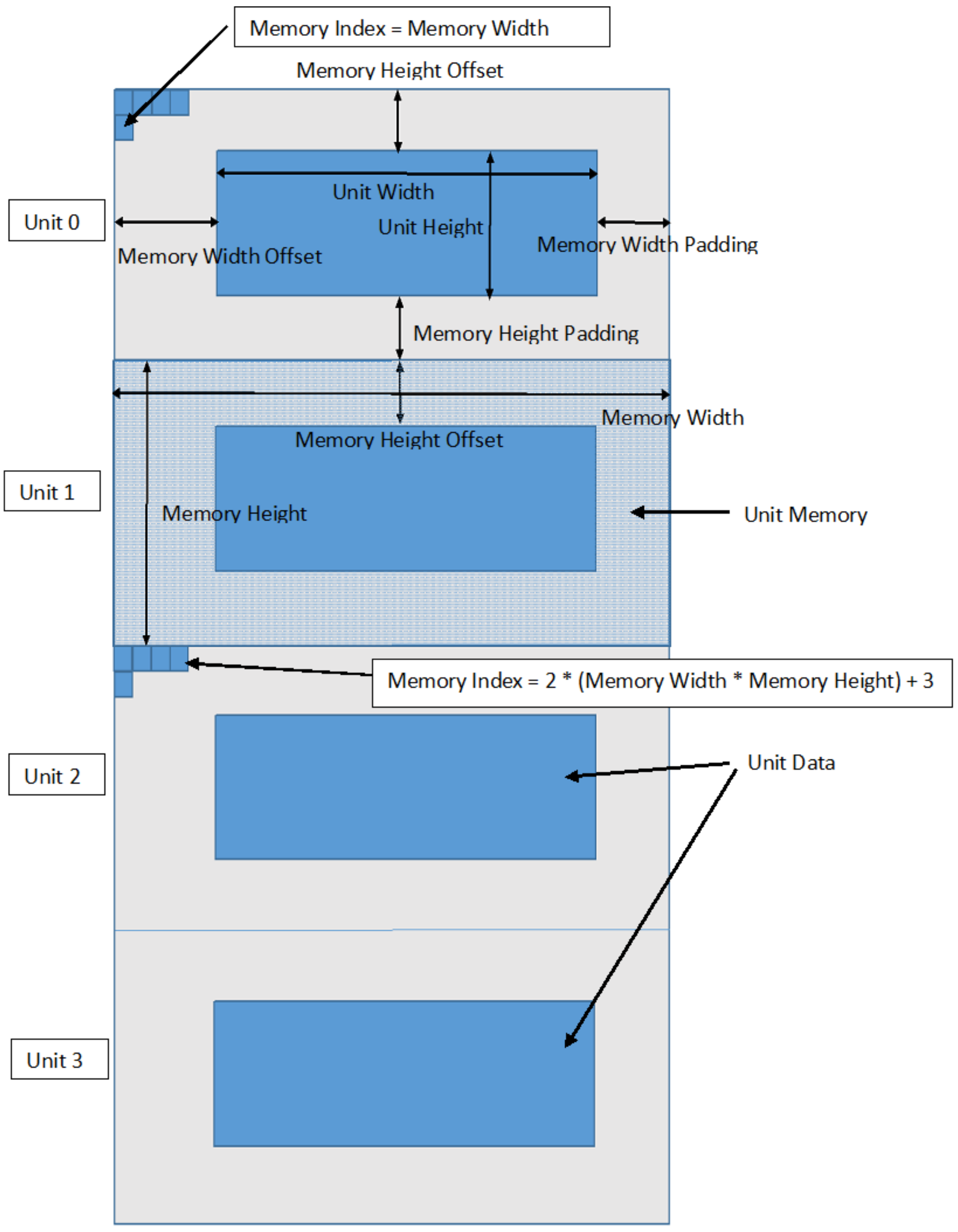

Figure 1: Image showing the memory layout of a BLOB that is shared between layers. This memory layout will allow for maximum flexibility for the layers to try find an optimized layout for their particular problem. Since we impose that every layer must handle this layout, we will also avoid unnecessary memory copies that may or may not require an additional kernel call.

Before deducing the equations needed for memory access, the following variable conventions are imposed when dealing with memory access:

Name	**Variable name**	**Macro name**
Unit offset	⟨description⟩UnitOffset	⟨DESCRIPTION⟩_UNIT_OFFSET
Number of units	⟨description⟩UnitCount	⟨DESCRIPTION⟩_UNIT_COUNT
Unit padding	⟨description⟩UnitPadding	⟨DESCRIPTION⟩_UNIT_PADDING
Number of memory units	⟨description⟩UnitMemoryCount	⟨DESCRIPTION⟩_UNIT_MEMORY_COUNT
Unit data width	⟨description⟩UnitWidth	⟨DESCRIPTION⟩_UNIT_WIDTH
Unit data height	⟨description⟩UnitHeight	⟨DESCRIPTION⟩_UNIT_HEIGHT
Unit data height * width	⟨description⟩UnitElements	⟨DESCRIPTION⟩_UNIT_ELEMENTS
Unit memory width	⟨description⟩UnitMemoryWidth	⟨DESCRIPTION⟩_UNIT_MEMORY_WIDTH
Unit memory height	⟨description⟩UnitMemoryHeight	⟨DESCRIPTION⟩_UNIT_MEMORY_HEIGHT
Unit memory height * width	⟨description⟩UnitMemoryElements	⟨DESCRIPTION⟩_UNIT_MEMORY_ELEMENTS
Unit memory width padding	⟨description⟩UnitMemoryWidthPadding	⟨DESCRIPTION⟩_UNIT_MEMORY_WIDTH_PADDING
Unit memory height padding	⟨description⟩UnitMemoryHeightPadding	⟨DESCRIPTION⟩_UNIT_MEMORY_HEIGHT_PADDING
Unit memory width offset	⟨description⟩UnitMemoryWidthOffset	⟨DESCRIPTION⟩_UNIT_MEMORY_WIDTH_OFFSET
Unit memory height offset	⟨description⟩UnitMemoryHeightOffset	⟨DESCRIPTION⟩_UNIT_MEMORY_HEIGHT_OFFSET

Table 1: A table showing the conventions for the variables describing the memory BLOB.

When accessing this memory inside a kernel, the actual data units and the data units used inside the kernel do not necessarily align. For example, when back-propagating from a perceptron layer to a convolution layer, the convolution layer would require that you have at least some padding around every unit so that the rotated convolution kernel may be applied directly. This means that the data width for the convolution layer, inside the kernel, is $2 * (filterDimension - 1)$ larger than the actual data width outside the kernel. The convolution layer must here assure that the offset / padding is set to zeros. Furthermore, some hardware could require that the amount of global work units is a multiple of 8 in order to use vectorization. In this case a sufficient amount of padding is added and some ”dummy” calculations are made on the padding.

In order to assure stability throughout the development of the kernels, we will also require the following conventions for global and local work units.

Name	**Variable name**	**Macro name**
Third global dimension size	globalUnits	GLOBAL_UNITS
Second global dimension size	globalHeight	GLOBAL_HEIGHT
First global dimension size	globalWidth	GLOBAL_WIDTH
Third local dimension size	localUnits	LOCAL_UNITS
Second local dimension size	localHeight	LOCAL_HEIGHT
First local dimension size	localWidth	LOCAL_WIDTH

Table 2: A table showing the conventions for the variables determining the global and local work sizes.

Every other variable defined needed for successful kernel execution should follow the above guidelines. The equations determining global memory access is given by:

$$x - widthIndex$$
$$y - heightIndex$$
$$z - unitIndex$$

$$\begin{gathered} index = x + \langle DESCRIPTION\rangle\_UNIT\_MEMORY\_WIDTH\_OFFSET + \\ \langle DESCRIPTION\rangle\_UNIT\_MEMORY\_WIDTH * (\\ \langle DESCRIPTION\rangle\_UNIT\_MEMORY\_HEIGHT\_OFFSET + y) + \\ \langle DESCRIPTION\rangle\_UNIT\_MEMORY\_ELEMENTS * (\\ \langle DESCRIPTION\rangle\_UNIT\_OFFSET + z) \end{gathered}$$

It's important to note that if you need to access an element inside a loop, you normally don't need to perform the entire calculating. It's enough to cache all the previous x,y,z values and simply perform a single addition inside the inner loop. Furthermore, for GPUs, the most important performance bottleneck lies in the memory access. This means that if you are using a memory access pattern that require a lot of arithmetic, but allows for better coalescing, then use it!