2016 2nd Workshop on Machine Learning in HPC Environments

Performance-Portable Autotuning of OpenCL Kernels for Convolutional Layers of Deep Neural Networks

Yaohung M. Tsai
University of Tennessee Knoxville
Email: ytsai2@vols.utk.edu

Piotr Luszczek
University of Tennessee Knoxville
Email: luszczek@icl.utk.edu

Jakub Kurzak
University of Tennessee Knoxville
Email: kurzak@icl.utk.edu

Jack Dongarra
University of Tennessee Knoxville
and Oak Ridge National Laboratory
Email: dongarra@icl.utk.eduu

***Abstract*—We present a portable and highly-optimized Deep Neural Network (DNN) algorithm and its implementation techniques. Our approach is a novel combination of existing HPC techniques that methodically applies autotuning as well as data layout and low-level optimizations that achieve performance matching and/or exceeding what is possible with either reverse engineering and manual assembly coding or proprietary vendor libraries. The former was done inside the maxDNN implementation and the latter is represented by cuDNN. Our work may be directly applied to the most time consuming part of DNN workflow, namely the training process which often needs a restart when it stagnates due to, for example, diminishing gradients and getting stuck in local minima. With the result of performance tests on a consumer-grade GPU with the latest High Bandwidth Memory (HBM) stack, our methodology can match a server grade hardware at a fraction of the price. Another tuning sweep on a new GPU architecture from a different vendor also attests to the portability of our approach and the quality of our implementation.**

1. Introduction

Machine Learning with Deep Neural Networks (DNN) has undergone a tremendous growth since the early days with projects such as DistBelief[1] that sought to harness the power of distributed computing. DNNs may be used to solve many machine learning tasks[2]. However, it was the pioneering work and the performance brought by GPUs and the DNN design called AlexNet[3] that changed the applicability and practical appeal of the methods. The new hardware allowed for drastically improved training time which now could be done in a week even for more than ten neural network layers. In fact, all of the entries for the ILSVRC challenge[4] now feature GPUs.

The Berkeley Caffe project[5] takes as input a DNN design in JSON format and automates the process of training and evaluation of the resulting neural network. NVIDIA's GPUs currently enjoy support for Deep Learning through the Caffe[5], cuDNN[6] and maxDNN[7] projects. Caffe includes direct calls to cuBLAS, cuDNN implements convolutional kernels natively, and maxDNN uses reverse engineered binary opcodes targeting Maxwell GPUs. Support for AMD GPUs is much less complete and our current work explores the feasibility of the Fury X card for Deep Learning and our results affirm not only the suitability of the AMD's first High Bandwidth Memory hardware but prove it to be competitive when used with the optimization techniques we propose in this paper. However, without a methodology for development of machine learning algorithms the resulting implementation might become highly suspect[8]. We seek to avoid problems of this kind by proposing an approach that takes an existing kernel for a dense matrix or tensor operation that demonstrates high efficiency on an OpenCL-enabled accelerator and turn it into HPC DNN with generic design that can serve a variety of deep learning tasks and network designs.

2. Contributions and Relevance

The contributions of this paper are as follows:
1) We use HPC techniques to bring deep learning to AMD accelerators through the use of OpenCL. The achieved performance levels are competitive with what is reported in literature for the server level GPU cards from NVIDIA branded as Maxwell architecture.
2) We present the *methodology* of how to proceed from an HPC formulation of a dense matrix and/or tensor kernel and use it as a starting point to obtain a highly optimized and yet generic implementation of a Convolutional Neural Network kernel that can be used for a variety of neural networks to allow the machine learning expert both the flexibility and efficiency without the burden of low-level manual tuning. We call the resulting code a *local memory kernel*.
3) We use autotuning for the final stage of our methodology to remove the need to write and rewrite the optimized

978-1-5090-3882-4/16 $31.00 © 2016 IEEE
DOI 10.1109/MLHPC.2016.5

code manually and limit the performance of the final implementation of what can be conceived by the HPC expert. Instead, we parameterize our implementation and thus create a search space that we subsequently explore which results in generation of numerous kernel codes. As an outcome, we get a much broader range of kernels variants which are likely to be missed by a human expert but are optimal in terms of performance due to the right interaction between parameters, optimizing compiler, and hardware components. It is worth noting that this is very important for a high-level kernel language provided by OpenCL that lacks intrinsics (for example for wide data load instructions), special data types (such hardware specific vectors), etc. In fact, OpenCL kernel implementations are subject to very aggressive code optimizations that might counter the intentions expressed by the HPC developer.

4) As far as we know, we present the first HPC implementation that uses standard OpenCL rather than NVIDIA CUDA or assembly/binary code that is common in deep learning community. This makes the OpenCL-enabled platforms available for HPC deep learning and provides the users with portability across a wider range of hardware while still training their neural networks with high efficiency.

5) We use as our implementation platform a consumer-grade GPU hardware from AMD – a few hundred dollar card, the first one to feature High Bandwidth Memory (HBM) stack that interfaces GPU compute units with the main memory through silicon interposers. The achieved results compare favorably with the high-end cards from NVIDIA such as Titan X programmed with lower-level code.

6) We focus only on the **single node performance** because this is still one of the main bottlenecks in large distributed deep learning networks. In fact, the gradient update may be reformulated to facilitate multi-GPU training and support so called *data parallelism*, *model parallelism*, and the hybrid approach[9]. The stochastic gradient descent method[10] allows to focus HPC effort at the single node level were the majority of performance is extracted through the methodology presented here and relies on communication between the nodes that resembles a *halo exchange* to scale the neural network training process to large distributed memory installations[11], [12], [13].

3. Background and Motivation

Convolutional Neural Networks (CNN)[14] are a machine learning technique well suited for classification and recognition tasks for images. As a supervised learning process, the input data first comes in and the neural network computes the outputs in the *feed-forward* stage. Because the *ground truth* outputs are known during training, the error is computed between the correct and computed outputs which is then used to *back-propagate* it back through the network. The training phase dominates the time spent on computational tasks of learning and classification/recognition and, consequently, Stochastic Gradient Descent (SGD) method was proposed to further speedup that phase and successfully applied to large data sets [15]. The basic idea of SGD is to compute the correction to weights based on a small batch of example inputs (images) and the details are beyond the scope of this writing.

TABLE 1. COMPARISON OF TERMS USED BY CUDA AND OPENCL TO DESCRIBE VERY SIMILAR CONCEPTS.

NVIDIA CUDA Term or Syntax	Khronos OpenCL Term or Syntax
GPU Hardware Components	
SM, SMX	CU
streaming multiprocessor	compute unit
scalar core	processing element (PE)
host thread	host program
thread block	work-group
thread	work item
grid	NDRange
shared (per-block) memory	local memory
local memory	private memory
texture cache	image
kernel	program
PTX†	IL‡
GPU Software Constructs	
`__global__ void K ()`	`__kernel void K ()`
`void K(float *X)`	`void K(__global float *X)`
`float *F;`	`__global float *F;`
`__shared__ float *B;`	`__local float *B;`
`int tx = threadIdx.x`	`int tx = get_local_id(0)`
`int bx = blockIdx.x`	`int bx = get_group_id(0)`
`__syncthreads()`	`barrier(CLK_LOCAL_MEM_FENCE)`

† PTX is Parallel Thread Execution

‡ IL is Intermediate Language

Convolutional Neural Networks (CNN) were inspired by the visual cortex of mammals [16] and are a variant of MLP with sparser connectivity that emphasizes spatial locality of perception. The current trend in the design of CNNs is to increase the number of layers and thus they are often called Deep Neural Networks (DNN). Further more, the layers do not have uniform size, connectivity, nor the same activation function. In fact, there exist very successful DNN designs that feature multiple activation functions (including sigmoid, ReLu, and max-pooling/subsampling), mix fully-connected and sparsely-connected layers, and drastically vary in the size and shape of the layers. This may be easily observed in the design of AlexNet [3] which won the ImageNet competition in 2012 and whose high-level design is shown in Fig. 1. Detailed analysis of this network is clearly beyond the scope of this writing but in summary, the layers in the network refine their specialization during training and become responsible for recognizing more abstract features the further from the left towards to the right the layer is located. Also, some of the layers on the left and closer to the inputs "learn" to act in a matter very similar to what can be achieved with widely used image processing algorithms such as edge detection or contour detection and the connectivity between these layers reflects the spatial locality of such methods. At the far right end of the network the signals communicated to the layers already contain information about higher abstraction features such as facial features and generic shapes of objects. Those are dealt with fully connected layers that operate based on principles of MLP [17].

Finally, we focus here on OpenCL concepts and code which have their straightforward equivalents in NVIDIA CUDA. Unfortunately, some of the terms and code constructs

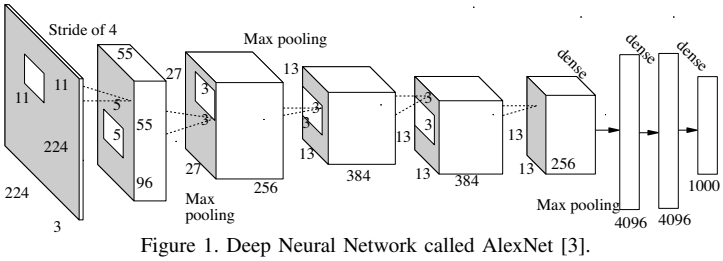

Figure 1. Deep Neural Network called AlexNet [3].

between the two might be contradictory when understood superficially. For this reason, we provide Table 1 to serve as a lookup glossary of terms and code snippets to guide the reader familiar with one of the technologies and not the other. More detailed analysis of the subject is beyond the scope of this writing and has been done by others[18], [19].

4. Related Work

Berkeley Caffe [5] is a deep learning framework developed by the Berkeley Vision and Learning Center. It's one of the most famous package because of it's performance and community support. At the early version, it was using the highly optimized single precision matrix-matrix multiplication (SGEMM) in cuBLAS [20] from NVIDIA to carry out convolution operations on GPU.

Due to the high demand, NVIDIA released their Deep Neural Network library (cuDNN) [6]. It contains CUDA kernels of commonly used DNN operations like pooling, forward and backward convolution. It may be used as a Caffe back end to speed up the training process.

Nervana Systems developed an assembler for NVIDIA Maxwell (MaxAs) [21] by reverse engineering the binary of Maxwell kernels. The single precision matrix-matrix multiply SGEMM kernel written by Nervana Systems using MaxAs is 5% faster than cuBLAS for certain input matrix sizes on the GTX980 card. MaxDNN [7] adopted the SGEMM kernel into the forward propagation operation. It reaches 96.3% of peak performance on the fifth layer of the Overfeat network [22] on GTX980. Nervana Systems released their deep learning framework and called it neon [23] with a full set of kernels developed with their assembler. The framework also supports half-precision floating-point arithmetic (FP16) which lowers required bandwidth without loosing much accuracy [24], [25], [26].

An active area of research is the use the Fast Fourier Transform (FFT) to compute the convolution. The main difference from the direct approach is that the algorithmic complexity is substantially lowered with FFT. There are known examples of efficient implementations of this approach for DNNs [27]. We do not use this technique in this work because it differs in critical algorithmic and implementation aspects such as: the work group (thread block) and loop structure, data layout, and padding of the intermediate data objects. Our goal is to compare against direct approaches that were used for established network designs [22], [28]. Furthermore, striding is a much more serious consideration with FFT-based convolution due to sparsification of the computed data and a need for pruned FFTs as opposed to the dense FFT especially for the left-hand-side layers that are close to the input.

Finally, it is obviously possible to compute the convolutions by following the mathematical formula directly. This is in fact the approach pursued by the older network designs such as cuda-convnet2[1] [9]. While the publicly available implementations use GPUs and achieve acceptable level of performance, the main drawback is a large optimization space required for optimal code that works well across the varying shapes of neural layers. The relative sizes of input images, filters, and output images change between the initial, middle, and final convolutional layers of the network. The image batch size also plays an important role and performance often drops as the number of images in a batch drops below a few tens of images [6]. The programming burden associated with optimization and then retuning, when the network design changes, is often too high to justify the effort to get the performance that can be achieved with other methods, especially with our *local memory kernel* approach that we present below.

The most recent research in Deep Learning focuses on using recursive formulation of the matrix-matrix multiply formulation that reduces the number of operations in the convolutions of the CNN layers. One such example is the use of the Strassen formula in CNN [29]. Another one is the use of the Winograd algorithm [30]. In general, the techniques arising from the arithmetic complexity theory may also be applied to further reduce computational complexity of the code at the algorithmic level and reduce the computation time (but not necessarily increase the performance).

5. Algorithm and Implementation Details

This section explains our implementation of the *local memory* convolutional kernel for AMD GPUs using OpenCL and introduces the BEAST autotuning framework and the tuning procedure itself that depends on both software and hardware restrictions called *constraints*, and, finally, the optimization based on characteristics of the computationally demanding convolution layers.

5.1. GEMM based convolution

The convolutional layer can be viewed as an operation which takes two 4D tensor as inputs and the output is another

1. https://github.com/akrizhevsky/cuda-convnet2

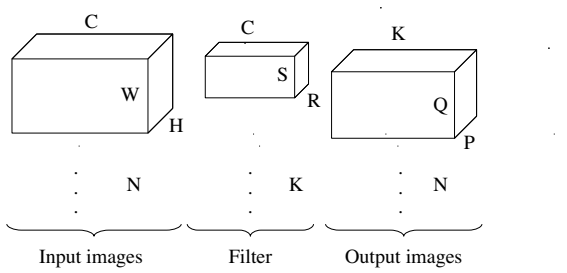

Figure 2. Input images, filters, and output images

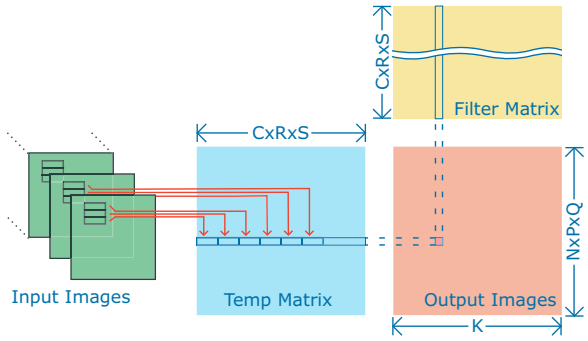

Figure 3. The `Im2col` approach from Caffe that uses cuBLAS before cuDNN was available.

4D tensor as well. This is shown in Fig. 2 with 3D batches of 2D images repeated N times and 3D filter batches with C-sized features repeated K times. In other words, the input images are aggregated as $I \in \mathbb{R}^{N\times C\times H\times W}$, where N is the batch size, C is the input feature size, H and W are the input image's width and height (the input images are scaled to a unified size before processing by CNN). The filter operation is also represented by a 4D tensor: $F \in \mathbb{R}^{C\times K\times R\times S}$, where K is the output feature size and R, S are the height and weight of the filter. The output images $O \in \mathbb{R}^{N\times K\times P\times Q}$ with P and Q as the size of a single output image. Each output feature $k \in [0,K)$ of an output image $n \in [0,N)$ is computed by summing the 2D convolutions between input image and filters:

$$O^{(n)}(k) = \Sigma_c \texttt{conv2}\left(I^{(n)}(c), F^{(c)}(k)\right) \quad (1)$$

By the definition of 2D convolution, the output element with index (p,q) where $p \in [0,P)$, $q \in [0,Q)$ is computed by the following:

$$O^{(n)}_{p,q}(k) = \Sigma_{c=0}^{C-1}\Sigma_{r=0}^{R-1}\Sigma_{s=0}^{S-1}\left(I^{(n)}_{p+r,w+s}(c) \times F^{(c)}_{R-1-r,S-1-s}(k)\right) \quad (2)$$

The inner-most triple nested loop that computes the above tensor product is similar in structure to a general matrix-matrix multiplication (GEMM) if the data layout is enforced and data duplication is performed if necessary. This leads to the so called `Im2col` approach used in Caffe [5] and other deep learning frameworks such as Torch [31] and Theano [32]. It casts each image into columns of the matrix, then calls highly optimized BLAS routines [33], [20], [34] for

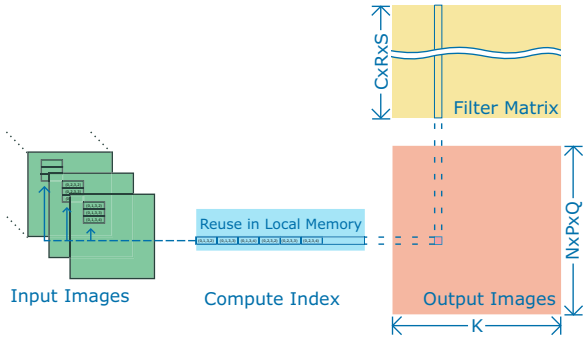

Figure 4. New convolutional kernel optimized for local memory reuse.

the best performance through vendor libraries. Fig. 3 shows a more detailed view of the `Im2col` approach whereby the input images on the left are copied into the Temp Matrix of size $C \times R \times S$ by $N \times P \times Q$ in the middle. A single input image will have to be repeated in the Temp Matrix for correct operation when passed to the underlying BLAS library. This repetition is done at least 9 times for a 3-by-3 filter and can be much higher for larger filters and for the *back-propagate* stage. Temp Matrix has to fit in the device memory of the GPU which limits the size of each individual image and the number of images in a batch. Even though an optimized matrix multiply routine is called, the potential performance benefit is limited because the same image is repeated in the Temp Matrix multiple times and has to be moved through the memory hierarchy multiple times. Also the work might have to be divided into several small single precision matrix multiplication calls to fit into the limited GPU memory, which would also reduce performance due to diminished parallelism and greater kernel launch overhead.

Fig. 4 shows our idea of a new *local memory* convolutional kernel that tackles the key problems associated with the `Im2col` approach. There is no longer duplication of input images in the device memory and OpenCL local memory is used more efficiently. It also removes the size limitations imposed on the the device memory of the GPU. In particular, with much lower memory requirement it is now possible to train the network with larger image batch sizes which speeds up the process and/or allows an earlier discovery of a failed convergence, e.g., due to learning stagnation, that may happen because of suboptimal value of the learning rate. The bandwidth usage is lowered by creating a single kernel that eliminates the extra device memory loads/stores when compared with `Im2col` that uses SGEMM kernels. This required us to change the data layout which precludes the use of standard library interfaces for matrix operations. Instead, for the purpose of training CNNs, we designed, implemented, and autotuned a new GPU kernel: the *local memory* convolutional kernel.

There are two extra sets of parameters originating in convolution layers: padding and stride. Padding is used to expand the input image to valid size by adding zeros around the boundaries in order to compute correct outputs when the filter is centered close to the edge and a part of the filter

must extend outside of the image – a common technique to compute a convolution without branch divergence. Finally, padding is required when computing back-propagation to update the filter. Stride is the displacement of the filter between consecutive outputs within a single image. Stride is usually used in early layers of the network to quickly lower the output image size while retaining the essential information. The use of stride would also make the FFT approach for these layers inefficient because large portion of the output will be discarded and thus would require truncated FFT that are hard compute efficiently. Note that "stride" in the context of this paper is different from its usual HPC meaning: CNN stride is the distance skipped in a layer of the network rather than HPC stride that is a distance skipped in the memory that often happens due to linearization of multidimensional arrays.

5.2. Implementation of the kernel

Figure 5 gives the comparison between the SGEMM kernel from clBLAS[34] and a example of our generated *local memory* convolution kernel for *forward-feed*. They share similar structure but our kernel has more integer arithmetic that computes the corresponding indices. More precisely, the index of $C \times R \times S$ dimension (in figure 3 and 4) is changing as the main loop index `k` increase. Integer modulo division is computed to get the updated (c, r, s) indices for checking the padding and the address offset for input image access. To minimize the performance impact from integer arithmetic, we feed the parameters from the layer as constants at compile time.

Another issue is the data layout. The most natural layout from the data management perspective is called NCHW for the order in which the dimensions are linearized in memory. This layout has been used in most of the packages because it makes it straightforward to read the image data from a file and pack them into GPU memory image by image. For example, NCHW is the default layout for cuDNN but CHWN is also supported with less performance, though. On the other hand, maxDNN prominently features the CHWN layout. It is worth noting that NCHW has several drawbacks from the perspective of writing GPU kernels where most of the performance is extracted. First, the dimensions of images might not be divisible by the hardware group size, resulting in non-coalesced memory access. Second, it is advantageous to have tunable parameters available inside the kernels' code for potential optimization and NCHW does not naturally lend itself to this technique. In our parametrized kernel code, we use as parameters the filter dimensions R and S. As a result, the compiler encounters these parameters as constants in the generated code which gives plenty of opportunity to use a variety of advanced compiler optimizations. Third, in addition to the computation time, it is important to consider the layout translation and mapping time that might dominate the execution under some circumstances. For example, when copying the input data and applying padding, the NCHW layout causes branch divergence between work items but with CHWN the whole work group would apply padding without the divergence because the HW dimension remains the same for all work items in the group.

For the *back-propagation* during the training process, the two kernels are implemented in the same manner, but they accept slightly different input data. The convolutional kernel propagates the error from output images and flipped filters and it computes the convolution which propagates the error one layer back (to the left). The *local memory* convolution kernel takes the errors from both layers and computes the convolution to update the weights in the filters.

5.3. Use of Autotuning

The Bench-testing environment for automated software tuning (BEAST) is an autotuning framework. According to the authors, "BEAST follows the classic recipe for automated software tuning. First, a computational kernel is implemented and parameterized with a set of tunable parameters (tiling sizes, implementation options, hardware switches), which define the search space. Then pruning constraints are applied to trim the search space to a manageable size. Then the variants that pass the pruning process are compiled, run and benchmarked, and the best performers are identified." [35] We used these tools to make our kernels suitable for autotuning and Figs. 7, 8, 9, and 10 are based on the obtained results which are described in Section 6. In order to fully explore the search space and find the optimal performance, our autotuning sweeps included parameters that turned out to be unsuitable for compilation, execution, or both. The space pruning takes place automatically and causes only a marginal overhead because the invalid kernels error out early. The benefit is that keeping the search space large allowed us to explore the expanded parameter set and avoid the problem with excessive pruning that occurs if even a single parameter is overly constrained. In a multidimensional search space pruning a single dimension automatically reduces the space in all other dimensions that are intertwined by construction.

5.4. GEMM search space

The GEMM tuning parameters have been studied on various GPUs on both CUDA [36] and OpenCL [37]. There were 5 major parameters picked here for our new *local memory* convolutional kernel. Fig. 6 shows the computation assigned to a single work group (thread block) in one iteration. Each work group is formed with M_{DIM} by N_{DIM} work items (threads). A sub-matrix of size M_{BLK} by N_{BLK} of output images matrix is going to be computed by a single work group. Each work item has M_{THD} by N_{THD} elements stored in their registers, with $M_{THD} = M_{BLK}/M_{DIM}$ and $N_{THD} = N_{BLK}/N_{DIM}$. At each iteration, two small matrices are constructed in local memory. One matrix is formed from input images with size M_{BLK} by K_{BLK} and the other one is from filters with size M_{BLK} by K_{BLK}. K_{BLK} is also the size of the inner loop of our *local memory* convolution kernel. A single work item will compute data indicated by dark blue rectangles.

```
for(int k=0; k<K; k+=16) {
  __local float* plA = lA + idx*97+idy;
  __local float* plB = lB + idx*97+idy;


  barrier(CLK_LOCAL_MEM_FENCE);

  //Load next submatrix into local memory









  for(int i=0; i<96; i+=16)
    plA[i] = A[i];
  for(int i=0; i<96; i+=16)
    plB[i] = B[i*ldb];




  barrier(CLK_LOCAL_MEM_FENCE);

  //Inner computation loop
  ...

  //Move to next submatrix
  A += 16*lda;
  B += 16;
}
```

```
for(int k=0; k<K; k+=16) {
  __local float* plA = lA + idx*97+idy;
  __local float* plB = lB + idx*97+idy;
  int x = p * stride_u + r;
  int y = q * stride_v + s;
  barrier(CLK_LOCAL_MEM_FENCE);

  //Load next submatrix into local memory
  //Check if it's in the padding region
  if( x < pad_h || x >= H+pad_h ||
      y < pad_w || y >= W+pad_w )
    for(int i=0; i<96; i+=16)
      plA[i] = 0;
  else
    for(int i=0; i<96; i+=16)
      plA[i]=A[index+c*N*H*W+r*N*W+s*N+i];
  for(int i=0; i<96; i+=16)
    plB[i] = B[i*ldb];

  //Update indices for next submatrix
  s += 16; r += s/S; s = s%S;
  c += r/R; r = r%R;
  barrier(CLK_LOCAL_MEM_FENCE);

  //Inner computation loop
  ...

  //Move to next submatrix
  A += 16*lda;
  B += 16;
}
```

Figure 5. Comparison between SGEMM kernel from clBLAS (left) and generated *local memory* convolution kernel (right).

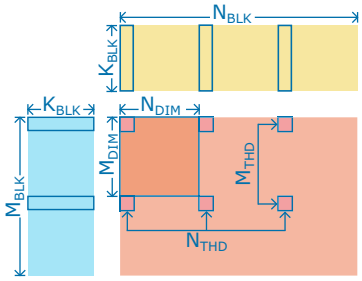

Figure 6. Autotuning Parameters for CNN transformed into matrix-matrix multiplication.

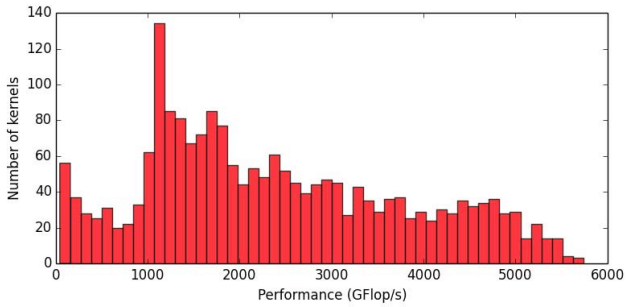

Figure 7. The histogram of performances from 2056 kernels on Alexnet L2.

In order to limit our search space, all the constraints from the algorithm, software, and hardware levels have to be applied. At the algorithm level, M_{BLK} has to be multiple of M_{DIM}; M_{BLK} has to be multiple of M_{DIM}, and K_{BLK} has to be multiple of both M_{DIM} and N_{DIM} to ensure that we don't need any boundary check in our main loop. For the software level, parameters from the convolutional layer are also taken into consideration. For example, in the forward-feed, we would pick M_{BLK} which can divide N and N_{BLK} can divide K so that the output images can be divided into work groups perfectly.

Assigning a variable to either SGPR (scalar general purpose register) or VGPR (vector general purpose register) is controlled by the compiler (sometimes called register coloring). Yet, the total number of needed registers is the responsibility of the programmer by the use of local variables and it should not exceed the hardware limit because it will lower the occupancy: we imposed a minimum of 2 active work groups per CU (compute unit) as a constraint to maintain a sufficient performance level and to prune the search space accordingly.

AMD APP Profiler (part of AMD CodeXL suite) can be used to reveal low-level parameters of the hardware that might not otherwise be available from the card specification. For the tested Fury X card, we observed 102 SGPRs , 256 VGPRs , and LDS (local data store corresponding to shared memory in CUDA) size of 32768. All these numbers are reported per CU.

6. Performance Results

The tested GPU was the AMD Fury X with peak single-precision (FP32) performance of 8602 Gflop/s and core frequency of 1050 MHz. AMD Radeon R9 Fury X is the

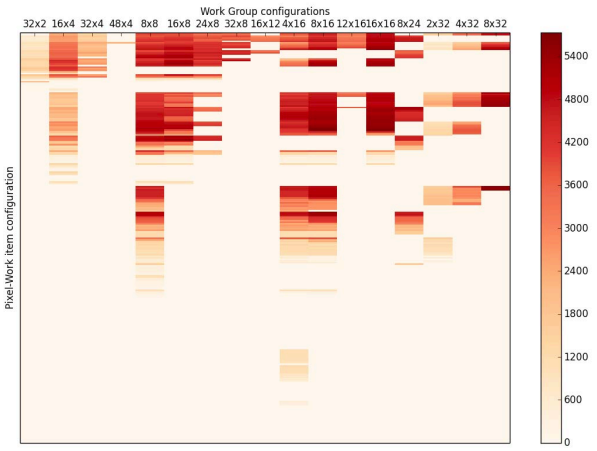

Figure 8. The tuning result of 4 parameters $(M_{DIM} \times N_{DIM}) \times (M_{THR} \times N_{THR})$ on Alexnet L2.

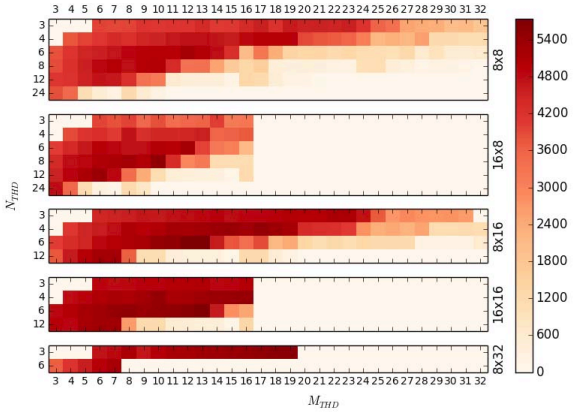

Figure 9. The tuning result of 5 work group configurations on AlexNet L2.

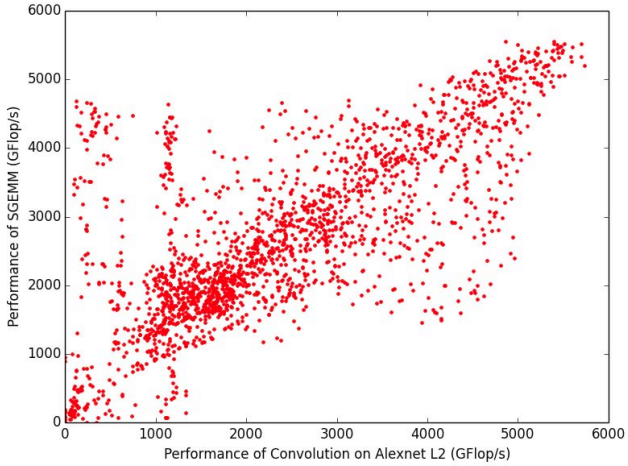

Figure 10. The performance comparison between Convolution on Alexnet L2 and SGEMM.

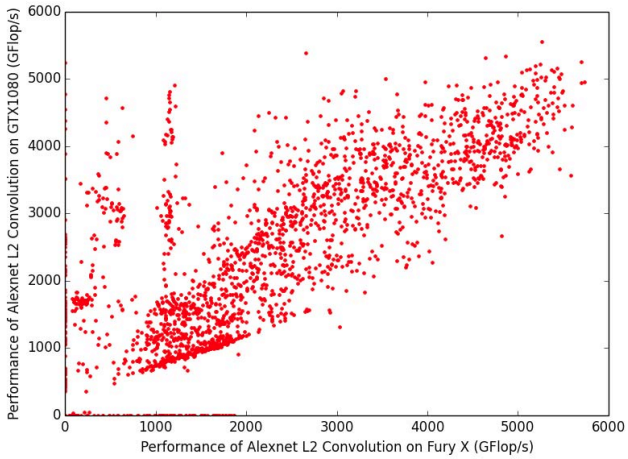

Figure 11. The performance comparison between Fury X and GTX 1080.

water-cooled high-end solution (compared with the basic Fury card which is air-cooled), which features 4096 GCN (Graphics Core Next) cores and 8.9 billion transistors with TDP of 275 W and the liquid cooler capable of dissipating up to 500 W (a potential for overclocking for even higher performance). Fury X features 4GiB of HBM memory – a 3D stacked memory that uses a silicon interposer. The memory takes up to 94% less space than the traditional design, is much faster, and has lower latency (due to being closer to the GPU), and much higher bandwidth of 512 GB/s. It runs at much lower frequency (500 MHz) through a 4096-bit memory interface.

Fig. 7 shows the distribution of the 2056 working kernels of AlexNet layer 2 (L2) across the performance range – other layers produced similar histograms. Performance ranged from 44.7 to 5735.8 Gflop/s. The best kernel achieved 66.7% of peak performance and 82 kernels are better than 5000 Gflop/s.

Fig. 8 gives an idea of how large the tuning space can be. The dimension of the heat map corresponds to 4 out of 5 tuning parameters: $(M_{DIM} \times N_{DIM}) \times (M_{THR} \times N_{THR})$. And for each heatmap block, the best result from different M_{BLK} (the 5th parameter) trials are picked. The figure has total of 17 columns of possible work group configurations $(M_{DIM} \times N_{DIM})$ and 254 rows of possible pixel assignment configurations $(M_{THR} \times N_{THR})$. 76.7% (3314) of the kernels were invalid and pruned by BEAST during the search space exploration and code generation. Only 3.4% (145) of the kernels failed at runtime due to compilation errors or hardware constraints. Both reasons contribute to the empty white areas in the heatmap. Nearly 20% (859) of the kernels were valid and are represented in the heatmap with the color corresponding to the achieved performance level. The darkest red indicates the best performance of 5735.8 Gflop/s and the closer to that shade of red a heatmap area is, the higher the performance was for the corresponding kernel.

The 5 heatmaps in Fig. 9 are taken from 5 columns of Fig. 8 and reshaped: each column becomes another 2D heatmap with the pixels assigned to configurations of M_{THR}

TABLE 2. THE PERFORMANCE OF CONVOLUTIONAL LAYERS IN ALEXNET WITH BATCH SIZE $N = 128$.

Alexnet	Forward feed		Back propagate (input)		Back propagate (filter)	
Neural layer	Performance (Gflop/s)	Time (ms)	Performance (Gflop/s)	Time (ms)	Performance (Gflop/s)	Time (ms)
L1	4972.0	4.5			4161.5	4.3
L2	5511.2	10.4	4795.5	12.0	2174.9	26.4
L3	5493.5	5.2	4936.4	5.8	3444.5	8.3
L4	5018.7	7.6	4878.5	7.8	3658.4	10.5
L5	4983.2	5.1	4964.6	5.2	3069.7	8.3
Combined Forward	5238.2	32.8	Combined Backward		3558.1	84.3

and N_{THR} on the x and y axes. The two best kernels here are from the same pixel assignments as 13×6 but with different work group configurations: 8×16 and 16×16.

Fig. 10 demonstrates the performance difference between our best *local memory* convolution kernel for AlexNet's second layer (L2) and the corresponding SGEMM kernel configuration. Each dot represents one set of tuning parameters. The x-axis and y-axis indicate the convolution kernel and SGEMM performance, respectively. There is visible correlation between the two. The Pearson correlation coefficient ρ is 0.808 ($\rho = 1$ implies ideal linear correlation). However, the configurations of the best convolution kernel achieving 5735.8 Gflop/s translate to only 5205.1 Gflop/s of SGEMM. On the other hand, the best SGEMM kernels at 5560.6 Gflop/s achieve 5394.8 Gflop/s for the convolution. In other words, the best SGEMM kernel configuration does not guarantee the best convolution kernel.The group in top left corner indicates that the parameters with sufficiently good performance ($\sim$4500 Gflop/s) for SGEMM may produce poor performance for convolution and achieve under 1000 Gflop/s.

Table 2 shows the performance of our convolutional kernels for the layers of AlexNet with batch size $N = 128$. The last column represents updating the filters, whose size ($K \times C \times R \times S$) usually is much smaller than either the input or the output images. Hence, there is a trade-off between occupancy and data reuse in local memory as the kernels have to pass the data between each other. The autotuning selects the best configuration but filter updates were still significantly slower than the kernels from the other two columns. It takes about one hour to autotune for single layer of the network. Compare that to the entire network training time, which could take weeks (repeatedly running the same set of kernels). The table may be used to compare our kernels to other libraries that run on NVIDIA Titan X [38] (comparisons against most codes favor our implementation by a large margin). Although we have not integrated our kernels into a deep learning framework, the time in other layers (ReLU and pooling) are usually around 10 ms and can be found in the Caffe output log. Our kernels show similar performance with other widely used optimized libraries by hardware or machine learning software vendors that use assembly (cuDNN) or binary code (maxDNN).

Finally, we would like to stress the performance portability of our methodology. Table 3 shows the tuning result from the recently released NVIDIA GTX1080 GPU based on the Pascal micro-architecture for each layer of Alexnet (*forward-feed*). The theoretical peak performance of GTX1080 is at 7967 Gflop/s – a close-enough match of Fury X's 8602 Gflop/s. Our autotuning approach gives us at least 57% of peak performance across all the layers on two totally different architectures. Fig. 11 also demonstrates the need for a kernel to be tuned for specific architecture. Each dot represents one set of tuning parameters. The x-axis and y-axis indicate performance on AMD Fury X and Nvidia GTX1080, respectively, for the *forward-feed* of Alexnet L2. Both are also highly correlated ($\rho = 0.792$), but in some cases, like the column of points around $x = 1000$, good performance may be achieved on GTX1080 but not on Fury X and vice versa.

TABLE 3. THE PORTABLE PERFORMANCE ACROSS LAYERS AND ARCHITECTURES.

Alexnet	AMD Fury X		Nvidia GTX1080	
Forward feed	Performance (Gflop/s)	% of peak	Performance (Gflop/s)	% of peak
L1	4972.0	57.8%	5279.2	66.3%
L2	5511.2	64.0%	5553.9	69.7%
L3	5493.5	63.9%	5595.8	70.2%
L4	5018.7	58.3%	5163.5	64.8%
L5	4983.2	57.9%	4732.5	59.4%

7. Conclusions and Future Work

In this paper, we proposed a methodology that allows systematic reformulation of simple dense matrix/tensor kernel written into a highly optimized CNN implementation in OpenCL. The resulting code is our new *local memory* convolution kernel for OpenCL-enabled GPUs that may serve as a very efficient implementation for a variety of convolutional layers in Deep Neural Network (DNN). Our *local memory* convolution kernel does not require extra temporary memory storage so that the limited memory of the GPU can be used more efficiently and a larger batch size N may be chosen for faster training of the network without a need to upgrade to a higher-end GPU with larger memory. We also showed that it is possible to implement a tunable kernel for convolution and achieve high performance without resorting to intricate coding practices and bogging down the code with lower-level details involving hardware architecture information and instruction-specific intrinsics. In fact, the code variants we generated were expressed in standard OpenCL that can be moved to new hardware and allow the local OpenCL compiler to generate platform-specific binary. The flexibility of our approach also allows to

tune specifically for each layer in the network because our approach permits different layer parameters and this results in different search space with narrowly targeted code for optimal performance. The performance portability across different layers and hardware architectures makes it possible to achieve good performance while trying a new network or system without putting extra efforts to optimize the code for it.

In the future, our next step would be to integrate our kernel generator and autotuner directly into a deep learning framework, so that the kernel can be automatically generated and tuned by only specify the CNN parameters at the level of network (rather than kernel) design by a machine learning expert. Also, the ReLU and pooling layers are usually used right after convolutional layers in modern DNN designs. This makes it possible to merge these layers into a single kernel to further lower the number of memory transactions and unnecessary data transfers between the levels of the memory hierarchy. Finally, yet another future direction is to investigate the complexity-reducing algorithms such as Winograd[30] for the layers with 3-by-3 filter size. Theoretically, this algorithm requires much less floating-point operations but it has to be implemented with care to limit the use of the register file.

Acknowledgments

This work was funded by NSF through Award Number 1439052. We would like to thank the BEAST team for help with autotuning sweeps.

References

[1] J. Dean, G. S. Corrado, R. Monga, K. Chen, M. Devin, Q. V. Le, M. Z. Mao, M. Ranzato, A. Senior, P. Tucker, K. Yang, and A. Y. Ng, "Large scale distributed deep networks," in *NIPS 2012: Neural Information Processing Systems*, December 2012.

[2] Y. Bengio, A. C. Courville, and P. Vincent, "Unsupervised feature learning and deep learning: A review and new perspectives," *CoRR*, vol. abs/1206.5538, 2012.

[3] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in Neural Information Processing Systems*, 2012.

[4] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C. Berg, and L. Fei-Fei, "Imagenet large scale visual recognition challenge," 2014.

[5] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell, "Caffe: Convolutional architecture for fast feature embedding," *arXiv preprint arXiv:1408.5093*, 2014.

[6] S. Chetlur, C. Woolley, P. Vandermersch, J. Cohen, J. Tran, B. Catanzaro, and E. Shelhamer, "cuDNN: Efficient primitives for deep learning," *CoRR*, vol. abs/1410.0759, 2014. [Online]. Available: http://arxiv.org/abs/1410.0759

[7] A. Lavin, "maxDNN: An efficient convolution kernel for deep learning with Maxwell GPUs," *CoRR*, vol. abs/1501.06633, 2015. [Online]. Available: http://arxiv.org/abs/1501.06633

[8] D. Sculley, G. Holt, D. Golovin, E. Davydov, T. Phillips, D. Ebner, V. Chaudhary, and M. Young, "Machine learning: The high interest credit card of technical debt," in *SE4ML: Software Engineering for Machine Learning (NIPS 2014 Workshop)*, 2014.

[9] A. Krizhevsky, "One weird trick for parallelizing convolutional neural networks," *CoRR*, vol. abs/1404.5997, 2014. [Online]. Available: http://arxiv.org/abs/1404.5997

[10] L. Bottou, "Stochastic gradient tricks," in *Neural Networks, Tricks of the Trade, Reloaded*, ser. Lecture Notes in Computer Science (LNCS 7700), G. Montavon, G. B. Orr, and K.-R. Müller, Eds. Springer, 2012, pp. 430–445. [Online]. Available: http://leon.bottou.org/papers/bottou-tricks-2012

[11] R. Wu, S. Yan, Y. Shan, Q. Dang, and G. Sun, "Deep image: Scaling up image recognition," *CoRR*, vol. abs/1501.02876, 2015. [Online]. Available: http://arxiv.org/abs/1501.02876

[12] J. Dean, G. S. Corrado, R. Monga, K. Chen, M. Devin, Q. V. Le, M. Z. Mao, M. Ranzato, A. Senior, P. Tucker, K. Yang, and A. Y. Ng, "Large scale distributed deep networks," in *NIPS 2012: Neural Information Processing Systems*, December 2012.

[13] A. Coates, B. Huval, T. Wang, D. J. Wu, A. Y. Ng, and B. Catanzaro, "Deep learning with COTS HPC systems," in *Proceedings of the 30th International Conference on Ma- chine Learning*, ser. JMLR: W&CP, vol. 28, 2013.

[14] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[15] L. Bottou and Y. LeCun, "On-line learning for very large data sets," *Appl. Stoch. Model. Bus. Ind.*, vol. 21, no. 2, 2005.

[16] D. Hubel and T. Wiesel, "Receptive fields and functional architecture of monkey striate cortex," *Journal of Physiology*, vol. 195, pp. 215–243, 1968.

[17] F. Rosenblatt, "The perceptron: A perceiving and recognizing automaton," Cornell Aeronautical Lab, Project PARA 85-460-1, 1957.

[18] P. Du, R. Weber, P. Luszczek, S. Tomov, G. Peterson, and J. Dongarra, "From CUDA to OpenCL: Towards a performance-portable solution for multi-platform GPU programming," *Parallel Comput.*, vol. 38, no. 8, pp. 391–407, Aug. 2012. [Online]. Available: http://dx.doi.org/10.1016/j.parco.2011.10.002

[19] M. J. Harvey and G. D. Fabritiis, "Swan: A tool for porting CUDA programs to OpenCL," *Computer Physics Communications*, vol. 182, no. 4, pp. 1093–1099, 2011. [Online]. Available: http://dx.doi.org/10.1016/j.cpc.2010.12.052

[20] "CUBLAS," 2016, available at http://docs.nvidia.com/cuda/cublas/.

[21] "Assembler for NVIDIA Maxwell architecture," Online https://github.com/NervanaSystems/maxas, 2015. [Online]. Available: https://github.com/NervanaSystems/maxas

[22] P. Sermanet, D. Eigen, X. Zhang, M. Mathieu, R. Fergus, and Y. LeCun, "Overfeat: Integrated recognition, localization and detection using convolutional networks," *CoRR*, vol. abs/1312.6229, 2013. [Online]. Available: http://arxiv.org/abs/1312.6229

[23] "Fast, scalable, easy-to-use Python based Deep Learning framework by Nervana," Online http://neon.nervanasys.com/, 2015. [Online]. Available: https://github.com/NervanaSystems/neon

[24] V. Vanhoucke, A. Senior, and M. Z. Mao, "Improving the speed of neural networks on CPUs," in *Deep Learning and Unsupervised Feature Learning Workshop, NIPS 2011*, 2011.

[25] M. Courbariaux, Y. Bengio, and J. David, "Low precision arithmetic for deep learning," *CoRR*, vol. abs/1412.7024, 2014. [Online]. Available: http://arxiv.org/abs/1412.7024

[26] S. Gupta, A. Agrawal, K. Gopalakrishnan, and P. Narayanan, "Deep learning with limited numerical precision," *CoRR*, vol. abs/1502.02551, 2015. [Online]. Available: http://arxiv.org/abs/1502.02551

[27] M. Mathieu, M. Henaff, and Y. LeCun, "Fast training of convolutional networks through ffts," *CoRR*, vol. abs/1312.5851, 2013. [Online]. Available: http://arxiv.org/abs/1312.5851

[28] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," *CoRR*, vol. abs/1409.4842, 2014. [Online]. Available: http://arxiv.org/abs/1409.4842

[29] J. Cong and B. Xiao, "Minimizing computation in convolutional neural networks," in *Artificial Neural Networks and Machine Learning*, ser. ICANN 2014. Springer, 2014, pp. 281–290.

[30] A. Lavin, "Fast algorithms for convolutional neural networks," *CoRR*, vol. abs/1509.09308, 2015. [Online]. Available: http://arxiv.org/abs/1509.09308

[31] R. Collobert, K. Kavukcuoglu, and C. Farabet, "Torch7: A MATLAB-like environment for machine learning," in *BigLearn, NIPS Workshop*, 2011.

[32] F. Bastien, P. Lamblin, R. Pascanu, J. Bergstra, I. J. Goodfellow, A. Bergeron, N. Bouchard, and Y. Bengio, "Theano: new features and speed improvements," Deep Learning and Unsupervised Feature Learning NIPS 2012 Workshop, 2012.

[33] "Intel math kernel library," https://software.intel.com/en-us/en-us/intel-mkl/.

[34] "clBLAS," 2016, available at https://github.com/clMathLibraries/clBLAS.

[35] P. Luszczek, M. Gates, J. Kurzak, A. Danalis, and J. Dongarra, "Search space generation and pruning system for autotuners," in *Submitted to IPDPSW*, ser. The Eleventh International Workshop on Automatic Performance Tuning (iWAPT) 2016. Chicago, IL, USA: IEEE, May 23rd 2016.

[36] J. Kurzak, S. Tomov, and J. Dongarra, "Autotuning GEMM kernels for the Fermi GPU," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 23, no. 11, pp. 2045–2057, 2012.

[37] K. Matsumoto, N. Nakasato, and S. G. Sedukhin, "Performance tuning of matrix multiplication in opencl on different GPUs and CPUs," in *High Performance Computing, Networking, Storage and Analysis (SCC), 2012 SC Companion:*. IEEE, 2012, pp. 396–405.

[38] "convnet-benchmarks," 2016, available at https://github.com/soumith/convnet-benchmarks.