From Matrix Multiplication to Deep Learning: Are We Tuning the Same Kernel?

Yaohung M. Tsai, Jakub Kurzak
Piotr Luszczek, Jack Dongarra

Innovative Computing Laboratory
University of Tennessee, Knoxville

Apr. 15, 2019
9th JLESC Workshop
Motivation

• Vendors provide low-level optimized libraries (Intel MKL, Nvidia cuBLAS/ cuDNN) can usually obtain near-peak performance.

• We still see rooms for improvement for special problem sizes, custom/ fused operations.

• Convolutional layers from deep learning were usually casted into GEMMs (GEneral Matrix Multiplication).

• But how similar are they? Can we apply the tuning result from one to the other?
Let’s Play with GPU GEMM, First

• Compute $C \leftarrow \alpha AB + \beta C$

• GPU GEMM kernel utilizes shared memory to reuse the data.

• Performance is coming from FMA (fused multiply–add) operations:

$$c \leftarrow a \times b + c$$
Search Space

- **Hardware level constraints:** Ensure there are enough registers and shared memory spaces with minimum occupancy.

- **Software level constraints:** Problem sizes are divisible by tile sizes to remove imbalance workload.

- **Algorithm level constraints:** Constraints from the kernel to remove boundary check and utilize all threads.
The goal of the BONSAI (Benchtesting OpeN Software Autotuning Infrastructure) project is to develop a software infrastructure for using parallel hybrid systems at any scale to carry out large, concurrent autotuning sweeps in order to dramatically accelerate the optimization process of computational kernels for GPU accelerators and many-core coprocessors.

We are developing a parallel, distributed benchmarking engine capable of scaling to tens of thousands of nodes, to benchmark millions of combinations of kernel configurations, problem sizes, and representative input datasets, while collecting hundreds of performance metrics such as time, energy consumption, cache misses, and memory bandwidth.

Compilation
BONSAI performs parallel compilation, both across distributed memory nodes and within each node. Compilation of a large number of kernels takes a significant time in the autotuning process. Numerical kernels are frequently heavily unrolled, which contributes to long compilation times. Also, compilation time can be extremely nonuniform. Therefore, BONSAI dynamically balances the workload.

Benchmarking
We use a standard MPI parallel job designed to work in existing batch queuing systems such as TORQUE PBS. This makes BONSAI deployable on a wide variety of systems, from small university clusters, to cloud computing, to national supercomputer centers. When available, BONSAI takes advantage of multiple accelerators within each node.

Data Collection
BONSAI will provide a framework to simplify the process of collecting hardware counters and performance data. We will leverage the various open-source and vendor-specific libraries such as NVIDIA's CUPTI API, AMD's CodeXL, Intel's VTune, and the open-source PAPI library. BONSAI will simplify the task of instrumenting the kernel and provide a simple interface for selecting the counters to be collected.

Data Analysis
We will provide a number of analytical tools and examples to guide the developer in analyzing their code. The analytical tools provided with BONSAI will include statistical and machine-learning tools in addition to a number of visualization utilities. These tools will leverage open-source data analysis libraries such as the PyData stack, R, and Spark tools such as MLlib.

Test Many Data Sets
In the course of our work on accelerating the ALS algorithm for collaborative filtering, we discovered that the optimal parameter configuration depends heavily on the properties of the input data set, which motivates tuning sweeps over many datasets. In particular, consider tuning a sparse matrix kernel by making a sweep over all matrices in the Univ. of Florida matrix collection (currently 2,833 problems and growing).

Test Many Data Layouts
Modern hardware is increasingly sensitive to the layout of data in memory. A number of different layouts have been proposed for dense linear algebra (row and column major, tile, space-filling curves), sparse linear algebra (CSC/CSR, ELLPACK, SELL-C/SELL-P, Sell-C-Sigma, BCSR, DIA, COO), deep learning (NCHW, NHWC), PDE discretizations (structure of arrays or array of structures), etc.

Collect Lots of Performance Metrics
NVIDIA Maxwell can collect 111 different hardware counter metrics, based on 75 different events, and only a few can be collected in a single run. This forces many reruns of the kernel to collect all relevant metrics. Similarly, Intel Xeon Phi Knights Landing can collect 119 native events, using 5 counters, also making it necessary to rerun the kernel multiple times.

LANAI: LANguage for Autotuning Infrastructure
R1 = range(3)
R2 = range(3)

@condition
def r1_smaller_than_r2():
    return R1 < R2

BONSAI Distributed Runtime
We are using a 4 nodes cluster with 4 Nvidia GTX1060s (Pascal) on each node for all the following results.

ST A1.3 BONSAI: Benchmarking OpeN Software Autotuning Infrastructure Jakub Kurzak
# GEMM Tuning Result

**Nvidia GTX 1060**  
Peak: 3855+ GFlop/s  
cuBLAS: 2980 GFlop/s

2945 Tuning points  
Done in 5 minutes!!

### Performance Histogram

![Performance Histogram](image)

<table>
<thead>
<tr>
<th>DIM_M</th>
<th>DIM_N</th>
<th>BLK_M</th>
<th>BLK_N</th>
<th>BLK_K</th>
<th>INDEX</th>
<th>Status</th>
<th>Error</th>
<th>Time</th>
<th>Gflops</th>
<th>Error</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compile</td>
<td>Compile</td>
<td>Compile</td>
<td>Compile</td>
<td>Compile</td>
<td>Compile</td>
<td>Compile</td>
<td>Output</td>
<td>Output</td>
<td>Output</td>
<td>Output</td>
</tr>
<tr>
<td>16</td>
<td>8</td>
<td>80</td>
<td>64</td>
<td>16</td>
<td>894</td>
<td>Success</td>
<td>None</td>
<td>0.002331</td>
<td>3174.07</td>
<td>3.69619E-07</td>
</tr>
<tr>
<td>16</td>
<td>8</td>
<td>64</td>
<td>64</td>
<td>16</td>
<td>892</td>
<td>Success</td>
<td>None</td>
<td>0.00239648</td>
<td>3087.34</td>
<td>3.69619E-07</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>72</td>
<td>64</td>
<td>8</td>
<td>532</td>
<td>Success</td>
<td>None</td>
<td>0.00242787</td>
<td>3047.43</td>
<td>3.69619E-07</td>
</tr>
<tr>
<td>32</td>
<td>8</td>
<td>128</td>
<td>64</td>
<td>32</td>
<td>1195</td>
<td>Success</td>
<td>None</td>
<td>0.00245839</td>
<td>3009.6</td>
<td>3.69619E-07</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>48</td>
<td>64</td>
<td>8</td>
<td>526</td>
<td>Success</td>
<td>None</td>
<td>0.00247354</td>
<td>2991.16</td>
<td>3.69619E-07</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>56</td>
<td>64</td>
<td>8</td>
<td>528</td>
<td>Success</td>
<td>None</td>
<td>0.00249576</td>
<td>2964.53</td>
<td>3.69619E-07</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>64</td>
<td>64</td>
<td>8</td>
<td>530</td>
<td>Success</td>
<td>None</td>
<td>0.00249576</td>
<td>2964.52</td>
<td>3.69619E-07</td>
</tr>
<tr>
<td>16</td>
<td>8</td>
<td>80</td>
<td>64</td>
<td>32</td>
<td>969</td>
<td>Success</td>
<td>None</td>
<td>0.00251015</td>
<td>2947.53</td>
<td></td>
</tr>
<tr>
<td>32</td>
<td>8</td>
<td>192</td>
<td>64</td>
<td>32</td>
<td>1199</td>
<td>Success</td>
<td>None</td>
<td>0.00254504</td>
<td>2907.13</td>
<td></td>
</tr>
<tr>
<td>32</td>
<td>8</td>
<td>224</td>
<td>64</td>
<td>32</td>
<td>1201</td>
<td>Success</td>
<td>None</td>
<td>0.00255048</td>
<td>2900.92</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>8</td>
<td>128</td>
<td>64</td>
<td>16</td>
<td>900</td>
<td>Success</td>
<td>None</td>
<td>0.00255777</td>
<td>2892.66</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>16</td>
<td>88</td>
<td>64</td>
<td>16</td>
<td>1725</td>
<td>Success</td>
<td>None</td>
<td>0.00256759</td>
<td>2881.59</td>
<td></td>
</tr>
</tbody>
</table>
Some Twist?

- Pairwise Euclidean Distance
- Common Operation in Machine Learning

\[ C(i, j) + = A(i, k) \times B(k, j) \]

\[ C(i, j) + = (A(i, k) - B(k, j))^2 \]
Are they still similar?

- Each dot represents a set of tuning parameters. x and y axis shows performance from 2 GPU kernels.

- There are cases that behave very differently between two kernels.

<table>
<thead>
<tr>
<th>DIM_M</th>
<th>DIM_N</th>
<th>BLK_M</th>
<th>BLK_N</th>
<th>BLK_K</th>
<th>INDEX</th>
<th>GEMM Gflop/s</th>
<th>DIST Gflop/s</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>16</td>
<td>176</td>
<td>64</td>
<td>16</td>
<td>2158</td>
<td>2810.79</td>
<td>3031.2</td>
</tr>
<tr>
<td>16</td>
<td>16</td>
<td>176</td>
<td>64</td>
<td>32</td>
<td>2234</td>
<td>2809.41</td>
<td>1616.88</td>
</tr>
</tbody>
</table>
Deep Learning: Background

Input layer         Hidden layer         Output layer

Input #1 →          →               →  Output
Input #2 →          →               →
Input #3 →          →               →
Input #4 →          →               →

Input Weight         Activation  Output

\[ y_1 = \sigma(\sum_{i=1}^{n} x_i \theta_i) \]
Alexnet for image classification.
Inference: Feed image into network and get the result likelihood vector.
Training: Back propagate the error and update the weights within layers.
Convolutional layers and fully connected layers are the compute intensive parts.

Discrete 2D Convolution

Output\( (x, y) = \sum_{i=0}^{r-1} \sum_{j=0}^{s-1} [\text{Input}(x - i, y - j) \times \text{Filter}(i, j)] \)
Dimensions of Convolutional Layer

- H x W 2D Input
- N: Batch size
- C: Input features
- P x Q 2D Output
- K: Output features
- R x S 2D Filter

\[
O^{(n)}(k) = \sum_c \text{conv2} \left( I^{(n)}(c), F^{(c)}(k) \right)
\]
GPU Kernel Implementation

- Construct part of the intermediate matrix in shared memory.

GEMM v.s. Convolutional Layer

```
for (kk = 0; kk < K; kk += blk_k) {
    //Load next submatrix of A into shared memory
    __syncthreads();
    //Load next submatrix of B into shared memory
    __syncthreads();
    //Inner computation loop
    __syncthreads();
    //Move to next submatrix
    A += blk_k*lda;
    B += blk_k;
}
```

```
for (kk = 0; kk < K; kk += blk_k) {
    //Load next part of Input into shared memory
    int x = p * stride_u + r;
    int y = q * stride_v + s;
    //Check if it's in the padding region
    if( x < pad_h || x >= H*pad_h )
        y < pad_w || y >= W*pad_w ){
        #pragma unroll
        for (n = 0; n < blk_k; n += dim_n_A){
            #pragma unroll
            for (m = 0; m < blk_m; m += dim_m_A){
                sA( n+idyA, m+idxA ) = A(m,n);
            }
        }
    } else {
        #pragma unroll
        for (n = 0; n < blk_k; n += dim_n_A){
            #pragma unroll
            for (m = 0; m < blk_m; m += dim_m_A) {
                sA( n+idyA, m+idxA ) = A(m,n);
            }
        }
    }
    //Load next part of Filter into shared memory
    __syncthreads();
    //Update indices for next submatrix
    s += 16; r += s/S; s = s%S;
    c += r/R; r = r%R;
    __syncthreads();
    //Inner computation loop
    __syncthreads();
    //Move to next submatrix
    A += blk_k*lda;
    B += blk_k;
}
```
Convolutional Layer Tuning Results

- Using one layer in ResNet: N=128 C=K=64 HxW = 56x56 RxS = 3x3
- The output size and number of Flops is the same as GEMM.

<table>
<thead>
<tr>
<th>DIM_M</th>
<th>DIM_N</th>
<th>BLK_M</th>
<th>BLK_N</th>
<th>BLK_K</th>
<th>INDEX</th>
<th>Status</th>
<th>Error</th>
<th>Time</th>
<th>GEMM</th>
<th>CONV</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integer</td>
<td>Integer</td>
<td>Integer</td>
<td>Integer</td>
<td>Integer</td>
<td>String</td>
<td>String</td>
<td>Real</td>
<td>Real</td>
<td>Real</td>
<td></td>
</tr>
<tr>
<td>Compile</td>
<td>Compile</td>
<td>Compile</td>
<td>Compile</td>
<td>Compile</td>
<td>Compile</td>
<td>Compile</td>
<td>Compile</td>
<td>Compile</td>
<td>Compile</td>
<td></td>
</tr>
<tr>
<td>32</td>
<td>8</td>
<td>256</td>
<td>64</td>
<td>32</td>
<td>1058</td>
<td>Success</td>
<td>None</td>
<td>0.0019924</td>
<td>2853.43</td>
<td>3713.49</td>
</tr>
<tr>
<td>32</td>
<td>8</td>
<td>224</td>
<td>64</td>
<td>32</td>
<td>1056</td>
<td>Success</td>
<td>None</td>
<td>0.00203969</td>
<td>2909.63</td>
<td>3627.39</td>
</tr>
<tr>
<td>32</td>
<td>8</td>
<td>192</td>
<td>64</td>
<td>32</td>
<td>1054</td>
<td>Success</td>
<td>None</td>
<td>0.00207288</td>
<td>2883.96</td>
<td>3569.31</td>
</tr>
<tr>
<td>16</td>
<td>8</td>
<td>112</td>
<td>64</td>
<td>16</td>
<td>821</td>
<td>Success</td>
<td>None</td>
<td>0.00215042</td>
<td>2830.04</td>
<td>3440.61</td>
</tr>
<tr>
<td>16</td>
<td>8</td>
<td>128</td>
<td>64</td>
<td>16</td>
<td>823</td>
<td>Success</td>
<td>None</td>
<td>0.00215454</td>
<td>2891.44</td>
<td>3434.03</td>
</tr>
<tr>
<td>16</td>
<td>8</td>
<td>80</td>
<td>64</td>
<td>16</td>
<td>817</td>
<td>Success</td>
<td>None</td>
<td>0.00216098</td>
<td>3167.65</td>
<td>3423.8</td>
</tr>
<tr>
<td>16</td>
<td>4</td>
<td>112</td>
<td>32</td>
<td>16</td>
<td>151</td>
<td>Success</td>
<td>None</td>
<td>0.00218383</td>
<td>1972.38</td>
<td>3387.97</td>
</tr>
<tr>
<td>16</td>
<td>8</td>
<td>80</td>
<td>64</td>
<td>32</td>
<td>682</td>
<td>Success</td>
<td>None</td>
<td>0.00221748</td>
<td>2931.88</td>
<td>3336.56</td>
</tr>
<tr>
<td>32</td>
<td>8</td>
<td>288</td>
<td>64</td>
<td>32</td>
<td>1060</td>
<td>Success</td>
<td>None</td>
<td>0.00222401</td>
<td>2582.34</td>
<td>3326.76</td>
</tr>
<tr>
<td>32</td>
<td>8</td>
<td>160</td>
<td>64</td>
<td>32</td>
<td>1052</td>
<td>Success</td>
<td>None</td>
<td>0.00223656</td>
<td>2644.34</td>
<td>3308.1</td>
</tr>
</tbody>
</table>

- Note: Winogrand convolution has better performance on 3x3 filter layers.
GEMM v.s. Convolutional Layer

- Same story: In general they are proportional to each other. But still some of kernels are good for convolutional layer but not for GEMM.
BOUNS: Pooling Layer

- Pooling layer: output maximum or average of the pooling window
- Similar memory read pattern
- Almost no computation
Lesson Learned and Future Work

- Modeling GPU kernel performance is hard.
- Models still could be used as guidelines for tuning sweep design.
- Improve our BONSAI framework.
- Machine learning model backend for BONSAI.
- This work is funded by NSF through Award Number 1642441.

ST A1.3 BONSAI: Benchmarking OpeN Software Autotuning Infrastructure Jakub Kurzak