Supplementary Material
DSC: Dense-Sparse Convolution for Vectorized Inference of Convolutional Neural Networks

Alexander Frickenstein
BMW Group
Autonomous Driving
alexander.frickenstein@bmw.de

Manoj Rohit Vemparala
BMW Group
Autonomous Driving
manoj-rohit.vemparala@bmw.de

Christian Unger
BMW Group
Autonomous Driving
christian.unger@bmw.de

Fatih Ayar
Technical University Munich
Electrical and Computer Engineering
fatih.ayar@tum.de

Walter Stechele
Technical University Munich
Electrical and Computer Engineering
walter.stechele@tum.de

S.1. Dense-Sparse Algorithm

The Dense-Sparse Convolution reduces the computational complexity of the convolutional layer by leveraging the Winograd minimal filtering algorithm and compression rate obtained from pruning. The overall algorithm is well demonstrated by Algorithm 1. The lines 1 to 7 separate the filter into sparse and dense, respectively. The storage format for sparse filter is used according to the CSR format. The sparse and dense output features are accumulated as finally result in line 23.

For sparse matrices, the direct sparse convolution is implemented. Weights are stored in the CSR format. The array `Data` stores remaining non zero elements, `col indices` stores columns indexed of each element and `row pointer` stores the number of elements in each row. Afterwards, convolution can be implemented as expressed in Algorithm 2.

S.2. Using SIMD Intrinsics

Data level parallelism is leveraged by SIMD instructions. SIMD instructions are Instruction Set Architecture (ISA) extensions which inform hardware to group data into vectors. SIMD instructions increase the arithmetic intensity of operations. Namely, more operations can be implemented with the same amount of data fetching. Similarly, since the single instruction is applied to multiple data, the total number of instructions to be fetched is diminished. However, the implementation of SIMD instructions requires low-level software knowledge and parallelism in the algorithm. Both, ARM NEON and Intel SSE extensions, provide vectorized arithmetic operations on 128-bit wide registers. SSE registers can store/load or multiply/add 8 full-precision `float` or 32 8-bit integer (`int8`) numbers. Even though the registers are mostly parallel in terms of basic operations, they might differ especially for low precision operations. Intel has an intrinsic operation, multiplying 8-bit unsigned and 8-bit signed numbers into an intermediate 16-bit signed number and then, adds adjacent pairs horizontally. In order to activate SIMD intrinsics, the code snippet in Algorithm 3 can be compiled as to activate SIMD intrinsics, i.e `g++ -msse3 foo.c`

In low-level programming executions are performed in the register level. Load and store operations should be hard-coded, which is different from high-level C/C++ coding. In order to be able to use a specific SIMD extensions, the corresponding header file should be added into the code and the corresponding compiler flag should be activated. For example, the code snippet given below fits four single precision inputs a and weights b into a 128-bit SIMD registers. The element-wise multiplication result is stored in C code.

Even though data vectorization can be implemented by low-level programming, modern compilers are also improved to provide *auto-vectorization*. In addition to the vectorization, other code optimization methods such as loop unrolling, inlining, pipelining can be performed by compilers. However, compiler optimizations do not guarantee vectorization for each instruction. Therefore, a better data level parallelism can be achieved by using low-level intrinsics. Furthermore, the

Algorithm 1 Dense-Sparse Convolutions.

Require: threshold to separate filter
1: **for** filter **do**
2: **if** number of nonzero elements [filter] <threshold **then**
3: $sparse\ filters \leftarrow filter$
4: **else**
5: $dense\ filters \leftarrow filter$
6: **end if**
7: **end for**
8: $CSR\ format \leftarrow sparse\ filters$
9: $filter\ format(data, num_filters, index) \leftarrow dense\ filters$
——————Sparse Filters
10: $sparse\ output \leftarrow direct\ sparse\ convolution(Algortihm2)$
——————Dense Filters
11: **for** c in $channels$ **do**
12: $\hat{g} \leftarrow GgG^T$ {Winograd transform of the input}
13: **end for**
14: **for** n in $out_channels$ **do**
15: $temp = 0$
16: **for** i in $num_fiters[n]$ **do**
17: **for** j in $index$ **do**
18: $temp+ = \hat{g} \odot (B^T dB)$
19: **end for**
20: **end for**
21: $dense\ output \leftarrow A^T[temp]A$
22: **end for**
—————Summation
23: $output \leftarrow sparse\ output + dense\ output$

Algorithm 2 Direct sparse convolution.

Require: Input feature,
Weights stored in CSR format as $Data$, $col_indices$, $row_pointer$,
Ensure: Output feature
for i in $row_pointer$ **do**
for j in range(row_pointer[i],row_pointer[i+1]) **do**
$coeff \leftarrow Data[j]$
$index \leftarrow col_indices[j]$
$output \leftarrow output + coeff \times input[index]$
end for
end for

structure of the algorithm can be shaped, based on vector implementation, in such way that a vectorized algorithm can boost the performance when it is implemented with the proper vector instructions.

S.3. Exploiting Thread Level Parallelism

After hitting the power wall in the semiconductor industry, the computational performance of CPUs has been increased by distributing computing over multiple cores on the same chip. This has two implementation aspects. The first aspect is that multicore design enables running different programs on different cores concurrently. The second aspect is that a single program can be divided into multiple branches. Multiple cores are placed into the same silicon die where each of the cores has their private L1 cache, whereas the L2 cache is shared.

Since the work is distributed among multiple cores, the theoretically achieved speedup is a multiple of the number of

Algorithm 3 Single precision floating-point MAC operation with intrinsics.

```
#include <immintrin.h>
int main(){
    //initialize aligned memory in the stack
    __declspec(align(16)) float a[4] = 1.0, 2.0, 3.0 , 4.0;
    __declspec(align(16)) float b[4] = 1.0, 2.0, 3.0 , 4.0;
    __declspec(align(16)) float c[4] = 1.0, 2.0, 3.0 , 4.0;
    //load data into vector registers
    __m128 vec_a = _mm_load_ps(a);
    __m128 vec_b = _mm_load_ps(b);
    __m128 vec_c = _mm_load_ps(b);
    //vec_c = vec_a * vec_b + vec_c
    //1.0+1.0*1.0, 2.0+2.0*2.0, 3.0+3.0*3.0, 4.0+4.0*4.0
    vec_c = _mm_fmadd_ps(vec_a,vec_b,vec_c);
    //store vector back into the memory
    _mm_store_ps(vec_c,c);      //c=2.0, 6.0, 12.0, 20.0
    return 0;
}
```

cores. However, due to the effort required for creating and merging threads, the theoretical speedup could never be reached. Furthermore, in general, not all parts of the workload can be parallelized. Some parts should stay sequential as the effort required for creating multiple branches exceeds the possible gain for small code snippets. The maximum acceleration by distributed computing can be calculated as Amdahl's law states:

$$Speedup = \frac{1}{1 - p + \frac{p}{n}}, \tag{1}$$

where p represents the parallel region and n represents the number of processors. Eq. 1 gives the maximum of the achievable speed-up. In practice, the obtained acceleration is even lower due to the effort spent on creating and merging threads. Moreover, the communication latency between threads also hurts the performance.

In a multicore CPU, typically each core has its own L1 cache and registers, whereas L2 and L3 caches are commonly shared. Thus, the communication between threads can be realized via shared variables through a common memory. That being said, multiple cores have access to reading and writing on the same variable. Hence, an extra communication protocol is not required. Although it is convenient to read variables from a shared memory, a cache coherency problem arises when multiple threads write on the same memory location. Therefore, parallelization on shared memory platforms should be implemented carefully in such a way that multiple threads never write on the same cache line simultaneously. That problem is called false sharing. False sharing hurts the performance of multithreading significantly because the local cache of each core must be updated from the main memory when a false sharing detected. In this section, we discuss the scalability of the Dense-Sparse Convolution on various CPUs.

S.3.1. Low-End CPUs

Low-end CPUs are available for low to moderate performance computers, such as Raspberry Pi 3 (RPI3) or casual personal computers. This kind of CPUs usually have 2-4 cores and 128-256 bit SIMD registers. Our Intel i5 CPU has two processors, however, it can handle four threads due to its hyper-threading capability. The RPI3 has four cores and can also handle four threads.

60% filter-wise sparsity is achieved on *conv3.2* layer of VGG16. Scalability of filter-wise convolution is measured against the scalability of dense convolution, as shown in Fig. 1. On the left, the implementation on RPi3 is illustrated. Filter-wise convolution can scale better than dense convolution with up to 3 threads. However, the performance decreases for four-threaded implementations. This is mainly due to the increasing power consumption of the RPi3. As the heat increases when consuming high power, the RPi3 has a mechanism for protecting its SoC by reducing the operational frequency of the chip. The other reason is that since the Operating System (OS) is running on one core when 4 threads are created, one of the threads has to share the core with the OS. Since the work is distributed equally over the cores, the remaining three cores have to wait for the slower one, reducing the overall performance.

Similarly, on the right side of Fig. 1, an implementation on the Intel CPU is shown. The Intel CPU has 2 cores but can handle 4 threads due to hyper-threading. Since multiple threads in a single core can share the L1 cache, a significant speed-up can be achieved by creating 2 threads. However, the third thread must be executed on a different core which has its own L1 cache and the communication between threads must be realized through the L2 cache. Therefore, the performance is either slightly increased or decreased between 2 and 3 threads. Even though the performance is remarkably influenced by the hardware, our filter-wise convolution algorithm is as scalable as full dense convolution.

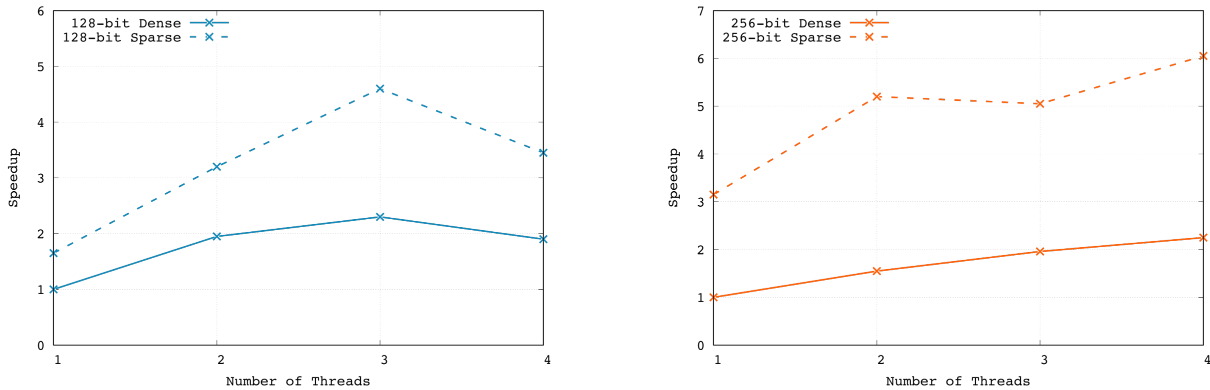

Figure 1. Scalability of Dense-Sparse Convolution on low-end CPUs. ARM-Cortex A53 (l). Intel i5-6300 (r)

S.3.2. High-End CPUs

High-end CPUs are preferred on servers or data centers where many independent operations are executed. Usually, they have more than 8 cores and larger cache sizes. The Intel Xeon CPU has 22 cores and can generate 44 threads. Moreover, it supports AVX512 SIMD extensions such that it has 512 bit wide SIMD registers. Therefore, the scalability is meassured of the dense and filter-wise convolutions in terms of data-level and thread-level parallelism, as illustrated in Fig. 2.

On data-level parallelism, we can observe that the performance increases proportionally to the width of the SIMD registers. That being said, the implementation with 256-bit SIMD registers is about $2\times$ faster and $2\times$ slower just as 512-bit registers. On thread-level parallelism, we have observed that the filter-wise convolution can scale up to 30 cores, as full dense convolution. After 30 threads, the performance drops mainly due to the saturation of the convolution operation. In other words, the computation is distributed over lots of threads such that the overhead for creating a thread becomes significant compared to the computation executed on the previous single thread.

In conclusion, whether it is a low-end or high-end CPU, our filter-wise sparse convolution algorithm can successfully be parallelized over many cores. A similar performance could be expected on GPUs since they share a relatively comparable architecture with multicore CPUs. Therefore, the advantages of pruning can be utilized even for fixed-hardware with regular sparsity and the suitable algorithm.

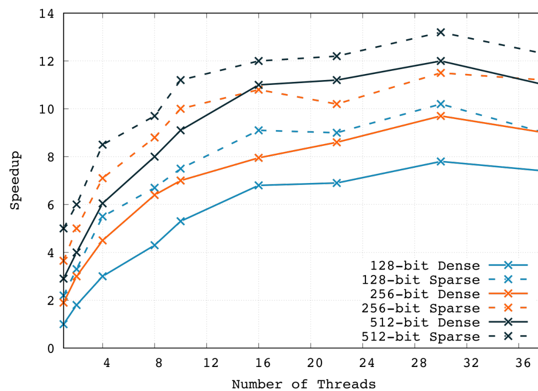

Figure 2. Performance comparison with varying threads and SIMD register sizes.Intel Xeon Gold 6152.