Revisiting POPCOUNT Operations in CPUs/GPUs

Chenfan Sun (Student) and Carlo del Mundo (Advisor)

University of Washington
{chenfs, cdel}@cs.washington.edu

ABSTRACT

Popcount is a binary operation where the input is a binary word and the output is the number of set bits. Popcount is a common building block for many applications such as the Hamming distance calculation. Recently, popcount is used to approximate multiplications in convolutional neural networks. Considering renewed interest in popcount, this work asks the question: can programmers lazily use the builtin popcount intrinsic or is further tuning necessary to achieve peak performance?

In this work, we benchmark the efficacy of several popcount implementations on both the CPU and GPU to analyze their behaviors under different working set sizes. On the CPU, results show that for memory bound workloads, the builtin popcount compiler intrinsic is within 0.01% of the fastest hand-tuned implementations suggesting that no hand-tuning is required, while this gap is up to 60% in the compute bound scenario where hand-tuned implementations of popcount matter.

1. INTRODUCTION

Popcount performance is critical for many applications such as binary neural network computations [1]. This paper examines the efficiency for popcount operations on both CPUs and GPUs.

We leverage an existing benchmark for popcount operations on x86 platforms [2]. This benchmark takes as input a working set size, and the output is a 64-bit unsigned integer that counts the number of set bits in the working set. This CPU benchmark contains different implementations of popcount ranging from the LUT-based approach to bit twiddle hacks. Then, we ported a subset of these implementations onto the GPU using the CUDA programming language. Our experimental testbed is a single computing node consisting of an Intel i7-4790K Haswell CPU and an NVIDIA GeForce GTX TITAN X. The node is configured with 32 GB dual-channel 1333MHz DDR3. For CPU implementations of popcount, we benchmarked implementations that are well optimized for sequential performance. We used the OpenMP framework to multithread the workload.

In the GPU implementations, we fuse several optimizations: (1) vector access and vector computation, (2) popcount methods based on LUTs or bit-twiddles, and (3) reduction via shared memory or via intra-warp shuffling. For GPU reduction methods, none of the implementations fully reduce the result into a 64-bit result. To complete the final computation, we perform the final reduction on the CPU. We did not include the CPU reduction time as it is negligible relative to the GPU computing.

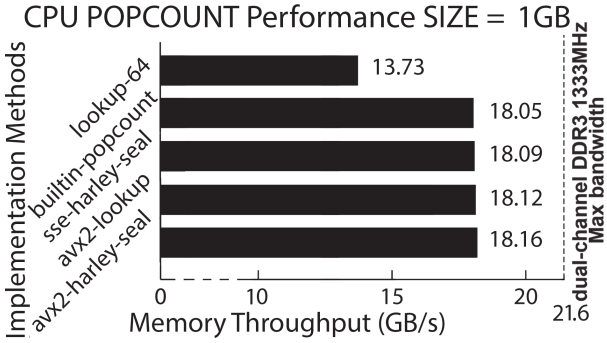

Figure 1: CPU Results (size = 1 GB). Since the dataset size is much larger than the CPU cache, the performance of popcount is memory bound. Hence, the builtin_popcount instruction is within 1% of the fastest versions.

We choose two dataset sizes, one fits in cache (size = 2MB), while the other exceeds the capacity of the cache (size = 1GB). These dataset sizes represent a compute bound and memory bound scenario, respectively. We run the test for 100 iterations and cumulate the runtime. We calculate the bandwidth by using the formula: $b =$ (SIZE * 100) / TIME, where SIZE is the dataset size, TIME is the cumulative execution time.

2. RESULTS

2.1 CPU

Figure 1 shows the performance results of various popcount implementations for a dataset size that exceeds the cache. Here, our working set size is 1GB. While, the best implementation is the Harley-Seal algorithm implemented in AVX-2, it is only within 0.01% faster than the builtin popcount intrinsic. Thus, for

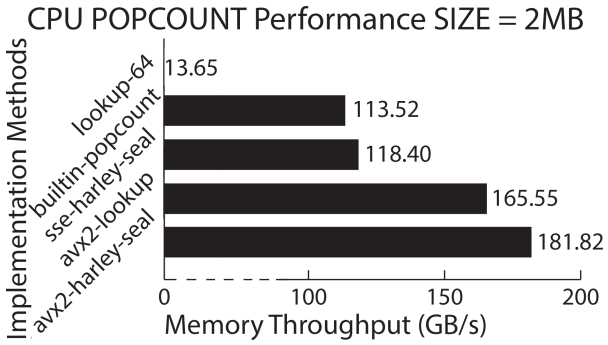

Figure 2: CPU Results (size = 2 MB). Since the dataset size fits with the cache, the performance of popcount is compute bound. The fastest hand-tuned implementation is up to 60% faster than the default builtin popcount instruction.

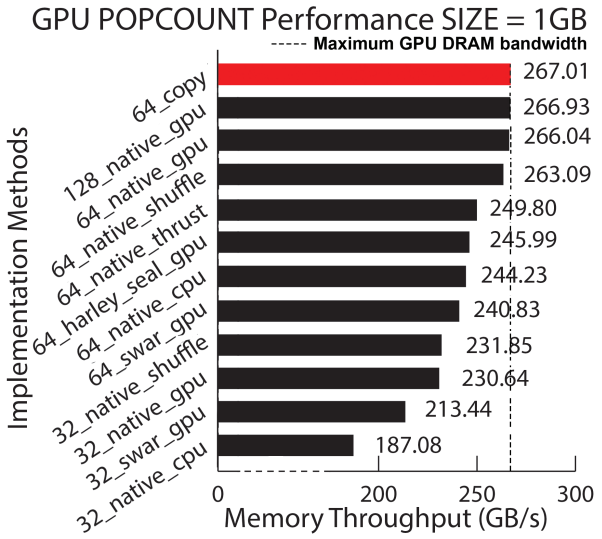

Figure 3: GPU Results (size = 1GB). Data widths improve increasing the performance. CUDA native popcount instruction outperform software level alternatives for popcount. Optimal popcount operations are dependent primarily by the reduction method. The fastest implementation is only 0.03% slower than CUDA SDK sample reduction code.

memory bound scenarios, using popcount intrinsic is sufficient.

Figure 2 shows the performance results for various popcount implementations for a dataset size that fits in cache. The best implementation of popcount is the Harley-Seal algorithm hand-tuned in AVX-2 vector extensions. Harley-Seal achieves 10% speedup than the next best performing popcount algorithm, and 60% faster than gcc's builtin popcount intrinsic.

One reason behind the gap between the builtin popcount intrinsic vs. the Harley-Seal algorithm lies in the efficient usage of CPU hardware resources. The Intel instructions manual suggests that the popcount intrinsic only uses one instruction port [3] out of the seven ports that Haswell offers. However, the Harley-Seal algorithm distributes the work amongst multiple execution ports by parallelizing the lookups across ports. Thus, the performance of Harley-Seal with AVX-2 is much higher than its hardware instruction counterpart.

The results for the CPU suggests that *for memory bound workloads, the popcount intrinsic will get you within 1% of the fastest hand-tuned implementation*, while the gap is at least 60% in the compute bound scenario. To improve the performance of popcount in memory bound scenarios, one must increase the bus bandwidth between DRAM and on-chip CPU memory. Increasing the number of memory channels, or using die-stacked memories [4] with their ample bandwidth can help alleviate this bottleneck.

2.2 GPU

Using the same dataset sizes, Figure 3 shows our GPU results. We only counted the computation time for executing the kernel, and did not include the data transfer time between host DRAM to GPU DRAM. On the y-axis, the leftmost number is the data width, and on the rightmost is the reduction method, and in the middle is the algorithm we used for computing the popcount. Native stands for the native intrinsic implementation of popcount on CUDA. The benchmark entitled "64_copy" performs a copy from GPU DRAM to on-chip and back to GPU DRAM to demonstrate the highest attainable bandwidth for reference.

Larger data widths incur better the performance, as vector data types such as 128-bit are better than their 64-bit counterparts which are better than their 32-bit counterparts. Reduction on the GPU is much faster than any other form of reduction (such as reductions on the CPU). Surprisingly, using the shuffle GPU intrinsic to perform the reduction within warps is much slower than performing a regular reduction using shared memory. We used the CUDA GPU reduction method from the CUDA SDK sample code. For popcount performance, the GPU popcount intrinsic outperformed the software backed bit-twiddling hacks. Optimal popcount operations on the GPU are dependent primarily by the reduction method. The use of the native popcount intrinsic is sufficient to compute the popcount in the GPU.

3. REFERENCES

[1] M. Rastegari, V. Ordonez, J. Redmon, and A. Farhadi, "XNOR-Net: ImageNet Classification Using Binary Convolutional Neural Networks," in ECCV 2016, Amsterdam, Netherlands.

[2] W. Mula, "SSSE3: fast popcount," Mar 2016. http://0x80.pl/articles/sse-popcount.html.

[3] A. Fog, "Instruction tables. Lists of instruction latencies, throughputs and micro-operation breakdowns for Intel, AMD and VIA CPUs," 2016.

[4] G. H. Loh, "3D-Stacked Memory Architectures for Multi-core Processors," in ISCA 2016, Beijing, China.