Evaluating the Jaccard-Tanimoto Index on Multi-Core Architectures

Vipin Sachdeva
IBM Future Technology and Solutions Design Center
Indianapolis, IN
vsachde@us.ibm.com

Douglas M. Freimuth
IBM Watson Research Center
Hawthorne, NY
dmfreim@us.ibm.com

Chris Mueller
Pervasive Technology Labs
Bloomington, IN
chemuell@cs.indiana.edu

***Abstract*— The Jaccard/Tanimoto coefficient is an important workload, used in a large variety of problems including drug design fingerprinting, clustering analysis, similarity web searching and image segmentation. This paper evaluates the Jaccard coefficient on the the Cell/B.E.™processor and the Intel ®Xeon ®dual-core platform. In our work, we have developed a novel parallel algorithm specially suited for the Cell/B.E. architecture for all-to-all Jaccard comparisons, that minimizes DMA transfers and reuses data in the local store. We show that our implementation on Cell/B.E. outperforms the implementations on comparable Intel platforms by 6-20X with full accuracy, and from 10-50X in reduced accuracy mode, depending on the size of the data. In addition to performance, we also discuss in detail our efforts to optimize our workload on both the Cell/B.E. and the Intel architectures and explain how avenues for optimization on each architecture are very different and vary from one architecture to another for our workload. Our work shows that the algorithms or kernels employed for the Jaccard coefficient calculation are heavily dependent on the traits of the target hardware.**

I. Introduction

Recent years have seen a resurgence in the number of hardware choices available to programmers. Multi-core processor architecture cores, which have multiple processing elements on a single chip are now the norm of the industry [1]. This has very important implications for many industries, which could now accelerate their workloads using these multi-core chips, and thus not relying only on single-thread performance. One such workload, which could benefit from multi-core technology is the Jaccard-Tanimoto index [2], which is is a correlation coefficient for determining the similarity between two binary strings, or bit vectors. Jaccard coefficient finds it's application in a wide variety of areas such as drug design [3], similarity searching on the Internet [4], financial applications and social network analysis [5]. The Jaccard-Tanimoto kernel for two vectors *x* and *y*, essentially consists of summing the bits in the *and* product and the *xor* product of the two vectors over their entire lengths, and dividing the bits set in the *and* product by this sum, in single precision. Mathematically,

$$Jaccard(x,y) = \frac{\textit{count\_bits(x and y)}}{\textit{count\_bits(x and y) + count\_bits(x xor y)}}$$

where *count_bits* specifies the number of bits set to 1 in the vector. We define the Jaccard workload for n vectors as computing and storing the Jaccard coefficient among each pair of these vectors.

II. Jaccard Index on the Cell/B.E. architecture

The Jaccard kernel can be broken up into 4 primary operations:

- Computing the *and* (*and_xy*) and the *xor* (*xor_xy*) product of the 2 vectors along their entire lengths, where *and_xy* and *xor_xy* are the vectors denoting the *and* and the *xor* product of the two vectors.
- Counting the bits set in *and_xy* and *xor_xy*, to obtain *bits_and_xy* and *bits_xor_xy*.
- Summing *bits_and_xy* and *bits_xor_xy* to obtain *bits_sum_xy*.
- Divide *bits_and_xy* by *bits_sum_xy*, and typecast the value as *float* to obtain *jaccard_xy*.

SPEs have instruction support for 128-bit *and* and *xor* products, which can be used for Step 1. Step 2 is the *pop-count* operation used in several important kernels, along with Jaccard coefficient; we use the *cntb* instruction for this step. The *cntb* instruction applies to 128-bit values, and for a 128-bit number *x* will return 16 8-bit values in the 128-bit result, each 8-bit value denoting the number of ones in the 8-bits of the input vector.

Our current implementation for summing the bits uses *sumb* instruction across two vectors (in our case *bits_and_xy* and *bits_xor_xy*), with appropriate shuffling using the *si_shufb* instruction. Using this approach, we are able to compute two vectors of type *vector short*, the last elements of which are *total_bits_and_xy* and *total_bits_xor_xy* respectively. The two vectors are then added using 128-bit *add* instruction to find *total_bits_sum_xy*. We then use the *cuflt* instruction, to change the *vector int* vectors to the *vector float* types. We extract the last elements of each of the vectors *total_bits_and_xy* and *total_bits_sum_xy* using the instruction *shli*, and then perform a scalar division, to return the result. This scalar division operation takes up more than 50% of the cost of the entire kernel in the SPE pipeline, excluding the DMA costs. We could also return the results to a lesser accuracy up to 12 bits, and then keep the entire kernel fully SIMDized: this could be implemented by computing the reciprocal estimate of the vector containing *total_bits_sum_xy*, and then multiplying it by the vector containing *total_bits_and_xy*, to find an approximated Jaccard coefficient. The advantage of this strategy is that the kernel is fully SIMDized, however at the expense of accuracy.

For data movement to and from SPUs, our parallel algorithm on Cell/B.E. for all-to-all comparisons finds a substantially optimal parallel solution through a runtime comparison of work allocated to every SPE. Given n vectors numbered 0 to $n-1$, we divide the Jaccard workload as follows: the vectors are divided amongst the SPEs by allocating $nvecs$ to each SPE in a round-robin fashion. A result vector of size $nvecs$ floating point values is also allocated in each of the SPU local store. When the first $nvecs$ (we call them *master vectors*) are DMAed to the SPE local store, the SPE computes the Jaccard coefficient within each pair of the $nvecs$ vectors. The result array for each i^{th} vector (comparisons from $(i+1)^{th}$ to $(nvecs-1)$ vectors) is stored to the PPU memory, before the computation for vector $i+1$ begins. Each of the result values of (i,j) jaccard comparison are actually 16-bytes to allow DMAs from any (i,j) value to the PPE memory. Before these all-to-all computations among $nvecs$ vectors are completed, another DMA request for the next $nvecs$ ($nvecs$ to $2*nvecs-1$ indices) are sent: we call them *slave vectors*. Once the Jaccard computation among every pair in the *master vectors* is completed, the Jaccard coefficient among this new set of $nvecs$ vectors (called *slave vectors*) and the master vectors is then initiated. This process of streaming in the $nvecs$ vectors (*slave vectors*) is repeated, until the entire set of n vectors in completed. Once the entire set of the slave vectors is completed, the next set of *master vectors* is streamed in; this process is continued until the end when all the computations are finished. Thus through this approach, we find the unique solution that maximizes reuse of the data streamed into the local store of the processing element. We perform all-to-all comparisons among every set of vectors that is streamed into the local store of the processing element, i.e. the master vectors (when they are streamed in), and then also between the master vectors and the slave vectors. This leads to maximal reuse of the data. In addition, our algorithm overlaps computation with communication to hide the latency of the data. Data is DMAed from main memory such that it arrives when comparisons among other data are being performed, e.g., when comparisons among master vectors are being performed, the slave vectors are being DMAed into the local store of the processing element. When one set of slave vector computations has started, the next set of slave vectors are already being streamed into the local store. A lower *nvecs* value denotes increased number of messages for streaming in the input vectors; a higher *nvecs* value leads to a lower number of messages but a less optimal load balancing strategy. Thus there are competing interests which are balanced by the runtime tests which select a $nvecs$ value that is optimum based on these considerations. In our experimental results, we vary the *nvecs* variable with the number of SPEs, to show the optimal number of *nvecs* variable for the number of SPEs. We show the detailed results of the implementation in the Section IV, which show that we achieve a super-linear speedup by optimizing several characteristics.

III. Jaccard coefficient on General-Purpose Processors

In the last few years, almost all the hardware manufactures including IBM, Intel and AMD have released multi-core general-purpose processors. For our implementation of Jaccard-Tanimoto, we decided to evaluate the Intel Xeon 5160, which is a dual-core processor. However, we cannot use the same kernel as the Cell architecture due to differences in instruction-set support. For computing the *and* and the *xor* product of the input vectors, we could use 128-bit SSE3 *_mm_and_si128* and *_mm_xor_si128* instructions to minimize the number of *and* and the *xor* operations. We would have to use each of these instructions only twice for computing *and* and the *xor* product of the input vectors (256 bits each). However, since there is no SSE3 instruction for pop-count, we would either have to map present scalar implementations for the pop-count operation to the SSE3, or use the general-purpose registers thereafter for the pop-count operation. This would incur some liability, as the results of the SSE3 computation will be transferred to the scalar registers. Algorithms for pop-count operation on general-purpose processors have been extensively researched, and we implemented several of these algorithms to find the most optimal approach. The fastest approach for the pop-count operation (as shown by runtime tests) on Xeon 5160, which leverages the caches on the Xeon, is to precompute the pop-count of unsigned numbers up to x bits, and store them in a table. We could then simply perform a lookup of the table at the index (equal to the number itself), and find out directly the number of bits set to 1, for the pop-count. The number x has to be chosen, considering that a fast access to the table is extremely important for this strategy; the Xeon 5160 has an individual 32 KB of L1 cache for each core, and a shared 4 MB L2 cache shared between two cores. It is important that the table fits into the L1 or the L2 cache, since we are looking at random indices, we do not expect the prefetching strategy to work well; for our experiments, we chose the table to work for unsigned numbers up to 16-bits, or $(2^{16}-1)$ elements, which is 65535 numbers. Each of these numbers are stored as type *char* datatypes, which makes the total size of the table close to 64 KB, which fits the table easily into the L2 cache. Thus, for the popcount operation for 256-bit vectors as in our input, using the table lookup, we will have to do 16 table lookups each for the *and* and the *xor* vectors. We could use the table-lookup approach with the 128 bit *and* and the *xor* operations for the entire Jaccard coefficient. On the other hand, we could also use the general-purpose registers for the *and* and the *xor* operation as well. We could do 64-bit *and* and *xor* operations, and then perform the pop-count using the table-lookup. It actually turns out that not using vector registers is infact faster, due to the overhead of moving data from vector registers to the scalar registers. We have manually unrolled the loops in using the 64-bit *and* and the *xor* operations, for optimal performance for the 256-bit vectors. Our implementation is parallelized by POSIX threads using a queue approach, and we report results for execution

on multiple cores of the Xeon 5160.

IV. Results

A. Experimental Setup

For both the Cell/B.E. and the Intel platforms, we present results varying the number of input vectors and the number of threads on each platform. The Intel Xeon 5160 was a two-processor system, with two Intel Xeon 5160 dual-core processors running at 3.0 Ghz connected through a 1.33 Ghz system bus. Each dual-core has a 4 MB L2 cache, and a 32 KB L1 cache for data and a 32 KB L1 for instructions. Our platform for the Cell/B.E. architecture uses the IBM BladeCenter QS21; QS21 has 2 Cell/B.E. processors running at 3.2 Ghz.

Our input datasets contain fingerprint bit vectors for 257271 of the compounds in the NCI compound database. The bit vectors are 166-bits long, zero padded to 256 bits. The bit vectors are stored as contiguous with 32 bytes per vector; thus the input dataset is more than 8 MB in size.

B. Results and Analysis

Table I shows the results for the Intel Xeon 5160, up to 4 threads. The results represent the times for the fastest Jaccard kernel explained in Section III; we time each of the kernels, and the time in Table I represent the lowest time of all the available kernels. Each of the threads run on a single-core: thus, 2 threads are running on a single Intel Xeon 5160 chip, and 4 threads on both the chips. The parallelization has been implemented with POSIX threads.

Table II shows the results for the QS21 platform, with number of SPEs varying from 1 to 16; the times shown represent the best values by varying *nvecs* variable. We time execution on each SPE varying the *nvecs* variable, and the time in Table II represent the lowest time for possible values of *nvecs*. The *nvecs* variable are varied from 16 to 1024 in our experiments, in multiples of 2. For lower number of SPEs, a higher *nvecs* gives the optimal result, as load-balancing is not of critical importance, but for higher number of SPEs, lower value of *nvecs* gives better results. Since each Cell/B.E. processor has 8 SPEs, 16 SPEs represents running the Jaccard coefficient on both the Cell/B.E. chips in the QS21. Comparing the results to the Intel platform, it is important we do a chip-to-chip comparison: thus, results for 4 threads on the Intel platform is equivalent to results for 16 SPEs on the Cell Broadband Engine. Table II shows the times for both the *full* and the *reduced* accuracy mode; more details on the reduced-accuracy mode were discussed in Section II. The fully SIMDized reduced-accuracy mode, actually is 2-4X faster than the one with the scalar division. This is due to two factors: the scalar division on the SPE is actually being performed by multiple instructions, and thus ends up taking a significant time of the total kernel. Also, with the reduced-accuracy mode, we keep the total kernel completely SIMDized as well.

Number of Input Vectors	Threads	Time (in seconds)
2048	1	49.76
	2	27.25
	4	21.13
4096	1	200.87
	2	105.59
	4	70.16
8192	1	832.21
	2	408.37
	4	257.87

TABLE I
PERFORMANCE OF INTEL XEON

Number of Input Vectors	Number of SPEs	Time (in seconds) (reduced-accuracy)	Time (in seconds) (full-accuracy)
2048	1	29.33	74.50
	2	16.37	40.25
	4	7.148	19.41
	8	1.481	7.96
	16	.242	0.815
4096	1	77.67	198.47
	2	42.846	107.61
	4	22.24	58.13
	8	10.58	27.13
	16	2.758	10.88
8192	1	320.814	801.18
	2	163.416	423.11
	4	89.15	219.66
	8	56.199	111.13
	16	23.12	42.85

TABLE II
PERFORMANCE OF CELL/B.E. ARCHITECTURE WITH VARYING NUMBER OF SPES

V. Conclusion

In this paper, we have evaluated the Jaccard workload on the Cell/B.E. and the Intel Xeon 5160 architecture. We have developed a novel parallel algorithm for the Cell Broadband Engine, that finds a substantially optimal parallel solution through a runtime comparison of work allocated to SPEs. The Cell/B.E. is shown to be up to 10X better in full accuracy, and up to 50X better in reduced accuracy mode over the comparable Intel platform.

Acknowledgment

References

[1] D. Geer, "Industry trends: Chip makers turn to multicore processors," *IEEE Computer*, vol. 38, no. 5, pp. 11–13, 2005.

[2] P. Jaccard, "The distribution of flora in the alpine zone," *New Phytologist*, vol. 11, pp. 37–50, 1912.

[3] P. Willett, "Similarity-based virtual screening using 2D fingerprints," *Drug Discovery Today*, vol. 11, pp. 1046–1053, 2006.

[4] T. Murata, "Machine discovery based on the co-occurence of references in search engine," in *Proc. Second Int'l Conference on Discovery Science*, Tokyo, Japan, Dec. 1999, pp. 220–229.

[5] D. Liben-Nowell and J. Kleinberg, "The link prediction problem for social networks," in *Proc. of the Twelfth Int'l Conf. on Active Media Technology*, New Orleans, LA, 2003, pp. 302–313.