The Changing Game of SIMD-optimized Joins

CS764 Final Report

The multiprocessor-optimized join has been a topic of academic interest since the 1980s. Optimization involves interfacing advances in both software and hardware to minimise runtime. One popular methodology is the use of Single Instruction Multiple Data intrinsics which enables the programmer to specify simultaneous (grouped) operations. New algorithmic joins designs, that are able to harness advantages of SIMD, have been suggested. This paper shows that these state of the art designs are limited in their applications and may sometimes return the same, or even negative results in comparison with standard join designs such as the Hash-based block nested loop.

Introduction

The intra-instruction parallel execution model of Single Instruction Multiple Data (SIMD) has been touted as a broadly applying method of obtaining speedup for generic problems including the relational join. For example, join algorithms, such as Hash and Sort-merge joins, typically involve some series of pairwise comparisons in their standard interpretation. By applying SIMD-optimizations, we can instruct the CPU to perform 4, 8, or 16 way group compare, hence at some level of abstraction, we may theoretically expect to obtain a 4, 8, 16x speedup in CPU time.

Implementers typically have two options to code SIMD optimizations in assembly or to utilize SSE intrinsics (Streaming SIMD instructions) which are "known by the complier and map to a sequence of one or more assembly language instructions. Hence they are inherently more efficient than called functions because no calling linkage is required"[1]. However in reality, implementers often struggle to obtain a linear speedup and to explain the source of their speedup since speedup can be swamped out by some factor x and produced by some other factor y. For example, loading and storing integers into SIMD registers requires the transformation into the __m128i or some other associated data type. Depending on the how concise the code is, and how much flexibility there is, loading and storing may also require some additional boilerplate. Hence while SIMD operations may be quicker than native operations, this speedup may be offset by the additional instructions or spatial unavailability. Particular domain constraints may also affect potential speedup. e.g. non-parallelizable code portions as represented under Amdahl's law.

Hence we believe that there is a disconnect between the generic notion of SIMD speedup and practical speedup for the specific database join problem that needs to be articulated to further the study of efficient join algorithms. The goal of this project is to examine more closely the source and potential between SIMD and non-SIMD join algorithm designs in previous work and execute some of them with a common set of experimentation variables to facilitate comparison.

This paper makes three main contributions. Firstly, we reimplement Zhou's SIMD-based nested loop designs and show that some of his assertions no longer hold. Secondly, we implement a hash-based version of Zhou's SIMD-based loop design and compare it to the non-SIMD optimized . Although neither SIMD design nor join design are new fields of study, to the best of our knowledge, there are no other implementations of SIMD-Hash-based block nested loop in published research. Thirdly, we present the findings from our own experimentation and assertions on the impact of the growing size of SIMD registers.

The rest of this paper is organized as follows. Section 2 covers related work in join algorithms. Section 3 details our implementation methodology (relevant SIMD operations, hardware system, data composition, join design), Section 4 describes selected experimentation (...), and Section 5 contains our conclusion and plans for future work.

Related Work

The most recent work in SIMD-optimized nested loops is (Zhou, 2004). This is the primary work that we base our assertions on and against. Although a tad old, it is a current work on SIMD loop designs. Loop designs are a good starting point because block nested loop is equivalent to simple hashing. This being the case, findings are likely to be valid in both domains. More recent work exist, but have focused on properties other than explicit SIMD-based design; e.g. (Spyros, 2011) examined the property of hash joins with growing main memory sizes.

Using a SIMD parallelism of four, Zhou asserted that his implementation resulted in a speedup of up to four times as compared to traditional algorithms. This reflects a linear speedup and so it appears that SIMD parallelism in loop design has been very effective. In other instances, a factor of 9 is even reported. While we do not doubt their results, we take issue with what constitutes a "traditional algorithm". Zhou does not discuss his baseline in very much detail, and the wording provides an indication that it is the more expensive N^2 nested loop rather than the cheaper and normatively implemented Hash-based block nested loop (H-BNL). We expect to see a lower speedup when comparing Zhou's design to a H-BNL baseline. However there is potential for improvement with a SIMD-optimized H-BNL implementation. We intend to develop such an implementation by simply adding a hash table to Zhou's design and to verify its relative

1http://msdn.microsoft.com/en-us/library/y0dh78ez(v=vs.71).aspx

speedup against a some variety of datasets. In doing so, we expect our experimentation to have a greater applicability and validity to different contexts. Another possible source of contention is Zhou's suggestion that the SIMD- optimized nested loop ran significantly faster because it avoided branch mispredictions which account for 40% of its time penalty in the N^2 nested loop case. Given that a significant number of years has passed since Zhou's paper was written, we explore whether this is still the case.

Lastly, we note that SIMD-designs for sort-merge exist (Chhugani, 2008; Kim 2009). Owing to practical constraints, we do not focus on sort-merge designs; However it is important to note that proponents of sort-merge have suggested that growing SIMD sizes will be the determining factor in sort-merge's ability to regain dominance in join processing. (Kim, 2009) states in bold print that "sort-merge join is projected to be 1.35X-1.65x faster than hash join". This claim is based on a supposed limitations of SIMD-optimized hash join, created by the need to divide records and updates amongst partitions and hash buckets and the cost of conflict detection. With this setup, they failed to obtain a positive speedup from their SIMD-optimized hash-join. In other words, they assert that the properties of SIMD and Hash join are like oil and water – they don't mix. However rather than a insurmountable design flaw, we see this as simply an implementation issue. The paradigm shift in (Spyros, 2011) was that hash join performs best in the no partitioning case. With no partitioning, hash joins would not suffer from the above-mentioned limitations, and we extrapolate that it would thus benefit greatly from SIMD optimization. Likewise, we expect to be able to overcome the stated limitation by modifying the hash table implementation or perhaps by limiting the required amount of partitioning that occurs e.g. partitioning either the hash table or the probe list only.

Algorithm Design

We base our experimentation on four algorithms:

1. N^2 nested loop
2. SIMD-optimized N^2 nested loop
3. H-BNL
4. SIMD-optimized H-BNL

Algorithm 1 and 2 are the algorithms implemented in (Zhou, 2009). Since no details are provided in (Zhou, 2009) about its N^2 nested loop, we implement it and also H-BNL according to their standard interpretations. The implementation for algorithm 2 follows the pseudo code provided in (Zhou, 2009) and is repeated here for clarity.

```
….
V = SIMD_bit_vector(mask);
if (V != 0) {
        for j= 1 to S {
                tmp = (V >> (S-j)) & 1;
                result[pos] = y[j];
                pos += tmp;
                }
        }
}
```

Algorithm 2: SIMD-optimized N^2 nested loop

Code samples of SIMD-optimized N^2 nested loop and H-BNL are listed in the Appendix.

Implementation Methodology

We implemented the join algorithms as describe earlier in a stand-alone C++ program. Most experimentation was done on a small sets of data on our laptop computer (Table 1). Occasionally, we also ran jobs on larger sets of data to verify our results (the same data used in (Spyros, 2011)) on a shared machine Intel Nehalem (Table 2) in the University of Wisconsin-Madison

	Intel(R) Core(TM)2 Duo
CPU	T9600 @ 2.80GHz
Cores	2
Cache	6144 KB, L2
Memory	3861 MB

Table 1: Platform characteristics of laptop computer

	Intel Nehalem
CPU	Xeon X5650 @ 2.67Ghz
Cores	6
Context per core	2
Cache Size, sharing	12MB, L3, shared
Memory	3* 4GB DDR3
Table 1: Platform characteristics of database server	

We do not report any join output times since we are primarily interested in reducing CPU time rather than I/O. My methodology was to first run configurations with output, checking the build table, checking the spilts, checking and collated results, and then to repeat the task without the output times.

Experimentation Results 1

	H-BNL	SIMD H-BNL
0.01_no.conf	77.0s	39.6s
0.01_.conf	3.2s	5.5s

0.01_no.conf and 0.01.conf use the same datasets for build and probe. However 0.01_no.conf is a no partitioning case, and uses a single thread, while 0.01.conf obeys independent partitioning and use a SMT of 4. Here we see that SIMD H-BNL performs as expected and is 1.94x faster than the standard H-BNL but performs more poorly under an independent partitioning assumption (0.01.conf).

Experimentation Results 2

The goal of this test was to verify that SIMD H-BNL provides a genuine improvement over the standard H-BNL. To verify this I test it on a much larger dataset located on the db1.chtc.wisc.edu server (the same basic dataset used in (Spyros, 2011)).

	Hash-based BNL	Hash-based SIND BNL
64 buckets	774s	63s
120 buckets	407s	52s

The configuration used is as follows with the only change being the number of buckets used. One reason for the significant jump in speedup is that when the size of the data grows, the minimizing cost of branch eliminations with group compares becomes dominant.

```
GNU nano 1.3.12     File: conf/64_server.conf
# vi:ts=2

path:   "datagen/";
bucksize:           1048576 ;

partitioner:
{
        build:
        {
                algorithm:          "independent";
                pagesize:            4194304;
                attribute:          1;
        };
        probe:
        {
                algorithm:          "independent";
```

```
                pagesize:           4194304;
                attribute:          2;
        };
        hash:
        {
                fn:                 "range";
                range:      [1,16777216];
                buckets:    64;
                skipbits: 17;
        };
};

build:
{
        file:   "dimension_01.tbl";
        schema: ("long", "long");
        jattr:  1;
        select: (2);
};
probe:
{
        file:       "fact_eq_100.tbl";
        schema: ("long", "long");
        jattr:  2;
        select: (1);
};

output: "test.tbl";

hash:
{
        fn:         "range";
        range:      [1,16777216];

        buckets:    64;
};
algorithm:
{
        copydata:       "yes";
        partitionbuild: "yes";
        buildpagesize:  32;
        partitionprobe: "no";
};

threads:            12;
```

How can we account for situations where the SIMD H-BNL performs more poorly?

In Zhou's paper, his SIMD implementations occurred almost no branch mis-prediction penalty while the standard nested loop occurred 40% miss-predictions in terms of elapsed time. This leads me to assume that his standard nested loop was poorly implemented. Assuming that the join selectivity is low, the number of bran miss-prediction in the standard nested loop can likewise be low (test !=0) although Branch miss-prediction A != Branch miss-prediction B. When we take away branch mis-prediction, the runtimes will look significantly more similar. Hence the performance of H-BNL can be the same as or even better than SIMD H-BNL, since there are also other factors to consider.

Cost breakdown:
SIMD N^2 nested loop = 2x Comparisons + Branch miss-prediction A+ Bit mask + <Write * 4
N^2 nested loop = 4x Comparison + Branch miss-prediction B , + Write
(Costs for H-BNL, SIMD H-BNL similar)

We see from the above cost breakdown that rather than 4x speedup, the result should typically be closer to 2x when we are considering these variables alone. We are ignore slight differences e.g. SIMD compare vs native compare speed.

A corollary result is that incurring a few branch miss-prediction is better than branch eliminations whenever the data is sizeable enough to detect a difference.
Cost breakdown (SIMD N^2 nested loop example) :
Branch miss-prediction heuristic = 2x Comparisons + Branch miss-prediction A + Bit mask + <Write * 4
Branch Elimination = 0x Comparisons + Branch miss-prediction B + Bit mask + (Write * Comparisons * 4)

Lastly one important point to note is that these results reflect a duplicate-inner design. (In the outer loop, fetch S join keys from the outer relation to make a SIMD unit. In the inner loop, fetch one join key from the inner relation and duplicate it S times to make another SIMD unit. Compare both units, check result, and product join accordingly). This incurs the cost of duplicating inner keys (cross product of relations / 4) times. When the data fits in memory, the cost of cache level instructions becomes significant. Implementing Duplicate-outer (the reverse of duplicate-inner) is trivial and will improve runtime, but left unfinished due to time constraints.

Conclusion

Given that the minimum speedup for SIMD loop joins is ~ close to 2x, we can determine without formal proof that it is unclear and unlikely that sort-merge will be faster than hash join (Kim's projection 1.35-1.65x) for any realistic dataset. Since SIMD H-BNL is very data dependent, there are constructed datasets underwhich it will perform very well or poorly. For example, to cause branch mis-predictions to occur on every iteration, simply locate a valid comparison within every set of 4 integers, or N integers depending on the N-way comparison size. Similarly, if the data is sorted, SIMD H-BNL is likely to perform very well, since matching comparisons and non-matching comparisons can be co-located respectively.

Future Work
Short-term: The presentation of experimentation in this paper only covers a pot-pourri of findings – more organized and detailed findings should be articulated time permitting.

Long-term: Given that there are circumstances and partitioning methods underwhich SIMD H-BNL may return same or negative results in comparison with H-BNL / sort-merge, more thought and design could go into optimizing Hash-based methods with SIMD instructions.

Appendix

SIMD-optimized N^2 nested loop

```
void NestedLoops::joinPageTupExperimental302(WriteTable* output, Page* page, void* tuple) {

int __attribute__ ((aligned(16))) vector1[4];
int __attribute__ ((aligned(16))) data2 = s2->asLong(tuple, ja2);
__m128i i2 = _mm_set1_epi32(data2);

int size3 = page->capacity() / page->tuplesize;
for (int i = 3; i < size3; i += 4) {
vector1[0] = page->getTupleOffset1(i-3);
vector1[1] = page->getTupleOffset1(i-2);
vector1[2] = page->getTupleOffset1(i-1);
vector1[3] = page->getTupleOffset1(i);

__m128i eq128 = _mm_cmpeq_epi32(_mm_load_si128(( __m128i *)vector1), i2);

int move = _mm_movemask_epi8(eq128);
if (move!=0){
int k = 0, m = 0;
int sout_size = sout->getTupleSize();
char tmp[sout_size*4];
for (int j = 0; j <= 12; j+=4) {
s1->copyTuple(&tmp[sout_size*k], sbuild->calcOffset(page->getTupleOffset(i-3+m), 1));
int sel2_size =  sel2.size();
for (int n = 0; n < sel2_size; ++n)
sout->writeData(&tmp[sout_size*k], // dest
s1->columns() + n, // col in output
s2->calcOffset(tuple, sel2[n])); // src for this col
k+= ((move >> j) & 1);
m++;
}
}
```

SIMD-optimized H-BNL

```
PageCursor* NestedLoops::probeNestedHash(SplitResult tin, int threadid, HashTable ht) {

WriteTable* ret = new WriteTable();
ret->init(sout, size);
HashTable::Iterator it = ht.createIterator();
PageCursor* t = (*tin)[threadid];
while (Page* b2 = t->readNext()) {
unsigned int i = 0;
while (void* tup2 = b2->getTupleOffset(i++)) { // probe // outer
unsigned int curbuc = _hashfn->hash(s2->asLong(tup2, ja2));
ht.placeIterator(it, curbuc);
// Duplicate Outer
unsigned int __attribute__ ((aligned(16))) data2 = s2->asLong(tup2, ja2); // probe
__m128i i2 = _mm_set1_epi32(data2);
unsigned int __attribute__ ((aligned(16))) vector1[4];

int sout_size = sout->getTupleSize();
char tmp[sout_size*4];

void *t124[4];
while (t124[0] = it.readnext()) { // iterate inner
t124[1] = it.readnext();
t124[2] = it.readnext();
t124[3] = it.readnext();

vector1[0] = sbuild->asLong(t124[0],0); // 0?
vector1[1] = sbuild->asLong(t124[1],0);
vector1[2] = sbuild->asLong(t124[2],0);
vector1[3] = sbuild->asLong(t124[3],0);

__m128i eq128 = _mm_cmpeq_epi32(_mm_load_si128(( __m128i *)vector1), i2);

int move = _mm_movemask_epi8(eq128);
if (move!=0){
int k = 0, m = 0;
for (int j = 0; j <= 12; j+=4) {
if (s1->getTupleSize())
s1->copyTuple(&tmp[sout_size*k], sbuild->calcOffset(t124[m], 1));
int sel2_size =  sel2.size();
for (int n = 0; n < sel2_size; ++n)
sout->writeData(&tmp[sout_size*k], // dest
s1->columns() + n, // col in output
s2->calcOffset(tup2, sel2[n])); // src for this col
k+= ((move >> j) & 1);
m++;
}
}
```