

# Parallelism in Database Operations

Kalle Kärkkäinen

**Abstract**—The developments in the memory and hard disk bandwidth latencies have made databases CPU bound. Recent studies have shown that this bottleneck can be helped with parallelism.

We give a survey of the methodologies that can be used to implement this kind of parallelism. Mainly, there are two kinds of parallel disciplines that have been discussed: 1) CPU parallel; and 2) GPU parallel.

The CPU parallel means employing the vector processor in the CPU. This is done by using certain machine code operations, that allows CPU to operate on multiple data items in the same instruction cycle. The GPU parallel means using the GPUs general programming interfaces to work on data in parallel.

We review recent research that has been done in both these areas of database parallelism.

**Index Terms**—Data processing, Databases, Database systems, Parallel algorithms, Parallel programming, Parallel processing.

## NOMENCLATURE

GPU	Graphics Processing Unit
GPGPU	General Programming GPU
CPU	Central Processing Unit
SISD	Single Instruction Single Data
SIMD	Single Instruction Multiple Data
SQL	Structured Query Language
IPC	Instructions Per Cycle

## I. INTRODUCTION

**S**PEED is but one of the design criteria of a database system. Reliability, conformance to the specifications and standards are often more desirable than speed.

In 1999 Ailamaki [1] et al. showed on average, database systems spend at least half of the operation time in stalls. They discovered that main causes for stalling were L2 cache misses and L1 instruction cache misses. Aside from those, 20 percent of the stalls were caused by branch mispredictions. All these areas can be helped with proper planning and implementation.

SIMD is a term coined by Flynn in 1966 [2]. It became part of Flynn’s taxonomy (see fig. I), a fourfold table with single data and multiple data on y-axis and single instruction and multiple instruction on x-axis. SIMD means that there is a single instruction set, that is applied to multiple data items. This type of parallelism is offered by vector processors and GPU.

Large databases have distinct problems, as the amount of data far exceed the capacity of the memory units, and therefore many layers of caching are necessary. Since the recent developments have made the latencies between caching layers radically smaller, new bottlenecks have appeared. As

	Single instruction	Multiple instructions
Single data	SISD	MISD
Multiple data	SIMD	MIMD

Fig. 1. Flynn’s taxonomy

discussed above, these bottlenecks are caused by program design problems.

Root cause of these problems becomes apparent as we form basic metrics, with which we can analyze the issue. As a computer operates a program, we can calculate the IPC performance metric. For a database systems this tends to be a low figure [1], [3]. This metric tells us that the database is not working optimally, and that it may be possible to improve the performance.

The hard core of a database system can be implemented as a virtual machine, able to run the SQL-queries. This type of implementation can be seen in the SQLite database. The virtual machine design enables one to reimplement the virtual machine in a new environment. Bakkum and Skadron implemented a subset of the SQLite virtual machine as a GPGPU program in 2010 [4].

Similarly, the different operations inside the SQL query can be implemented to make full use of the vector processing unit of the CPU. Zhou and Ross discussed in 2002 [5] that using CPU SIMD instructions may help to alleviate the branch mispredictions, and they implemented two systems, one using a sequential patterns and another using the SIMD instructions.

Both of these approaches try to tackle the bottleneck problem. On one hand the implementation problems are similar. Both require specialized hardware to be effective, which limits their usability for the general case. On the other hand GPU program design differs from the design of the CPU program. We try to analyze what considerations underlay each design.

### A. Outline

We first describe the problem of the low IPC count, starting with the internal working of a CPU and then describe the CPU and GPU solutions. Then we review the results given by the research papers, and finally present the common conclusions and discuss the differences in the approaches.

## II. BACKGROUND

The 10 000 ft picture of how CPU works is as follows: it loads an instruction from the systems memory, decodes the operation, executes the command, stores the result and moves to the next instruction. Each of these steps requires a CPU

```

x = [1, 2, 3, 4, 5]
S = 4;
N = 5;
condition = value >= 4

```

(a) Common parameters

```

for i = 1 to N {
  if (x[i] > 4) then
    result = result + x[i]
  // result is 0, 0, 0, 4, 9
  // in different iterations
}

```

(b) Sequential operation

```

for i = 1 to N step S {
  Mask[1..S] = SIMD_condition(x[i..i+S-1]);
  // mask is [0, 0, 0, 1], [1, 0, 0, 0]

  temp[1..S] = SIMD_AND(Mask[1..S], y[1..S]);
  // temp is [0, 0, 0, 4], [5, 0, 0, 0]

  sum[1..S] = SIMD_+(sum[1..S], temp[1..S]);
  // sum becomes [0, 0, 0, 4],
  // [5, 0, 0, 4]
}
aggregating sum = 9

```

(c) SIMD operation

Fig. 2. Example of addition as sequential and SIMD operation.

cycle and therefore each instruction requires certain amount of cycles to complete. A simple example of these steps is in a classic RISC processor:

- 1) Instruction fetch cycle (IF)
- 2) Instruction decode/Register fetch cycle (ID)
- 3) Execution/Effective address cycle (EX)
- 4) Memory access (MEM)
- 5) Write-back cycle (WB)

This would mean that each cycle, the CPU is able to process  $\frac{1}{5}$  of an instruction, each instruction being 5 cycles, causing CPU to use  $\frac{4}{5}$  of its time to process the 'plumbing'. To help with this problem, CPU's incorporate something called instruction level parallelism or *pipelining*. The CPU loads instructions during every cycle, and each of the instructions in the pipeline proceeds thru the stages at the same time. This brings cycles per instruction towards 1 [8].

Not all instructions are equal. A LOAD instruction requires 5 stages, but a BRANCH requires only 3 as it does not need to write anything back to memory, and because of this composition difference all applications have differing average cycles per instruction [3].

IPC is the reverse of cycles per instruction. It tells us how many instructions are run for each cycle. The differences in the operations allow the IPC to rise over 1, but this requires careful fencing inside the CPU, to ensure data integrity and correct operation. Those problems are out of scope for this paper, and are not covered at all.

#### A. Why the IPC metric is low in database systems

In a modern CPU the *pipeline length* could be 16 instructions, meaning that each cycle there are 16 instructions advancing thru the execution stages. This would lead us to believe that the expected IPC count for a system would be determined by

$$p = \text{pipeline length} \quad s = \text{average execution stages}$$

$$\frac{p}{s} = \text{average completing instructions per cycle}$$

The reality is more complex. As the CPU faces a branching statement such as IF ... THEN ... ELSE ..., it is forced to make a guess about the direction of the program flow. If the guess is wrong, pipelines need to be flushed and refilled with the correct branch. Branch mispredictions can cause significant delays in the processing [1], [3], [5].

In the database systems, branches are based on external data, and are impossible for the CPU to predict. Therefore branch mispredictions are very expensive for these types of applications, and the IPC count is lower than it should be [3].

Next we take a look at how the parallelism is achieved with the two different models and what kind of limitations they have. We start with the *vector* or *array processing* model, and then describe the GPU model.

#### B. The CPU solution: vector processing

In order to explain the parallelism achieved with vector processing methods, we need to take a look at how some basic instructions are executed, for example ADD instruction.

When CPU calculates basic addition, it uses its ADD instruction. These instructions come in many flavors, operating on different type of data or incorporating different overflow checking, but they all boil down to adding two figures in order to produce a third one. The figures have been loaded into registers, and are then added together and result is placed in a register from where it can be read after the instruction has completed.

The vectorized method is similar to the basic method. Only difference is that the registers involved contain multiple numbers. For example, CPU might support 128 bit vector registers, allowing 4 individual 32 bit floats to be calculated in the same cycle.

As we clearly see, vector processing has data parallelism equal to:

$$w = \text{width of the input register in bits}$$

$$b = \text{number space requirement in bits}$$

$$\frac{w}{b} = \text{numbers processed in parallel}$$

Vector processing applies the same instruction to all elements in the data, making it a SIMD implementation.

For other operations aside from addition, there is the special case of branching. Since the vector processor calculates on multiple numbers, IF branching is different from in sequential style. There are vector processing instructions for basic tests (equal, greater, lesser) that produce a result for all the input in one cycle. The output is a bit mask, 1's for matches, 0's for non-matches. The critical difference is that there is no JUMP directive in the matching, therefore there is no need for branch prediction [9].

If we take the addition as an example operation, we can verify that a SIMD addition over a group of numbers produces the same result as a similar algorithm using sequential addition.

In fig. 2 we show an addition algorithm with both SIMD operations and Single Instruction, Single Data (SISD) operations. For clarity and consistency, we show the operations at higher level in the figure as this has been the chosen method in the Zhou and Ross' study [5], and explain some of the assembly details here. We mainly focus on the SIMD in the explanation. The algorithms are from fig. 3 and fig. 5.

The sequential SISD algorithm is straight forward, and needs little explaining. It iterates from 1 to N, checks a condition for each iteration and builds an aggregate sum from each matching element. This simple algorithm works as a comparison baseline for the SIMD version.

When we analyze the SIMD version, we notice that the iteration now happens in chunks of data, the algorithm operates from 1 to N, in chunks of S. The algorithm computes SIMD\_condition, SIMD\_AND and finally SIMD\_+ operations for each chunk. Let us start from the top, and explain each operation in detail.

SIMD\_condition is a comparison operation, and in this case it compares the data chunk  $i \dots S-1$  to the comparison vector of all fours. The notation used in the listing is not obvious, but it follows the pattern set by Zhou and Ross [5]. Their basis for this notation was that the ICC compiler supports special high level constructs that allow writing a SISD operation, for example  $(4 \text{ ; } x) \&\& (x \text{ ; } 8)$ , as a similar SIMD operation, namely  $(\text{FOUR SIMD}_{<} x) \&\& (x \text{ SIMD}_{\leq} \text{EIGHT})$ , where FOUR and EIGHT are SIMD units containing S copies of 4 and 8. With this in mind, we can interpret the listing a little more.

The parameters given to SIMD\_condition are the two vectors of the size S. If we translate this into assembly, we have two registers, that point to memory locations that contain the values for the vectors. These memory locations are used to load data from memory into SIMD registers, which are then used by the SIMD comparison operation to compute the output, which contains bits set to 1 is the comparison is true for the same elements from the given vectors, and 0 otherwise.

The AND operation is a bitwise operation that takes two values, and produces a result that has a bit set if that bit was set for both of the values. The SIMD counterpart (SIMD\_AND) of this works in a similar fashion, taking two vectors as parameters and producing a third one, that contains their bitwise AND. In our example, SIMD\_AND operates on the

mask and the value vector. Since the mask is a bitfield containing either all 1 or 0 for the same index element of the value vector, the and operation suppresses all the values that did not match the earlier condition.

The SIMD\_+ operates on two vectors, producing their element wise addition. It adds the elements of the first vector to the same index element of the second vector. In our example we collect an index wise running sum of the vectors.

The sequential example has the result ready after all the elements are processed. In the SIMD version we ended up with a vector that contains the element wise sum. We can then aggregate each of those elements into one set of S results (the sum variable). Aggregation of result array into a single value can be with SIMD rotate and SIMD addition operations [5].

From this example alone we can see that these operations require the data elements to be placed strategically. As the SQL search query has criteria on certain data elements, and if the database system applies some of the strategies described here, the data layout must be columnar for the system to be the most efficient. If the layout is not columnar, the system needs to transform the dataset into a columnar shape if SIMD operations are to be used.

Vectorizing compilers, such as Intel's ICC, try to make use of the SIMD operations in some common operations [10]. [10] also describes that such generic compilers often fail to apply vectorization. Reasons for failing vary, from stylistic issues, hardware issues, and complexity issues [5].

A database system could make use of the vector processor in its core; processing many records at the same time. In section III-A we take a look at the findings from one such implementation.

### C. The GPGPU solution

GPU is designed to work on pixels and vertices, running operations on them written as shaders (pixel or vertex shader). GPU contains multiple processing cores, which simultaneously apply the shader transformation to each of the input pixels or vertices. The shaders are also known as *kernels*.

The GPU hardware is very limited in its functionality. Functionality missing includes branching (not a widely supported feature), locking (atomic primitives) and shared memory. The inherent design of the GPGPU program and the limitations of the hardware cause it to be effective in the subset of problems it is intended to be used in.

GPGPU is the generalized application of the same kernel idea. We have a stream of records, to which we apply the same operations. The stream processor offers data parallelism as the processor has multiple cores calculating different sections of the stream. All the processors use the same kernel for all the records that they process.

Much research has been done in the GPGPU area, and many GPU computation facilities are often used to do computationally intensive tasks. Many these tasks have similarities to classic database operations [6]. Research has emulated on the GPU to prove that it is a feasible prospect [7].

There are special drawbacks for the GPGPU solution that do not exist in the CPU solution. Memory needs to be copied

**Sequential:**

```

for i = 1 to N {
  if (condition(x[i])) then
    process1(y[i]);
  else process2(y[i]);
}

```

**SIMD:**

```

for i = 1 to N step S {
  Mask[1..S] = SIMD_condition(x[i..i+S-1]);
  SIMD_Process(Mask[1..S], y[i..i+S-1]);
}

```

Fig. 3. Basic scan-like algorithm as defined by Zhou and Ross.

over in most of the systems, leading into memory latency that takes away from the operational speed gains. This becomes a problem if datasets are large (lot to copy) and kernels small (little computation time per record), as in this setting the copy times become a ruling figure. In systems where the GPU memory is shared as part of the system memory, such as mobile computers, this type of copying is not necessary.

Another drawback for the GPU solution is the limited memory size. Many commercial GPU's operate with as little as 128 MB of video memory. In conjunction with the memory copy latency, this limitation may cause performance problems when database is large.

A database system could process the entire search in the GPU, streaming data records into GPU memory and processing the query as a kernel program. In section III-B we take a look at the findings of a GPGPU implementation of the SQLite virtual machine.

**III. REVIEW OF PAPERS**

In this section we take a look at papers that implemented and tested the two methodologies, and we review their results against the aforementioned background. First we take a look how SIMD on CPU affects the performance and then we take a look at the GPGPU solution.

**A. Implementing Database Operations Using SIMD Instructions**

Zhou and Ross described how the SQL operations may be implemented, and how those implementations would transform into SIMD applications [5]. They provided data for a wide variety of database operations, such as SELECT, GROUP BY, JOIN and database indexes. We will focus on part of their results that can be compared to the GPGPU paper, simple selects and aggregates.

In fig. 3 we can see the basic structure of the algorithm used by the paper. Their paper describes using a high level language for SIMD operations (they use Intel's ICC compiler), and as such their example pseudocode listings are at a higher level. The sequential algorithm processes each record at the time, testing it against a condition, and branching according to the test. The SIMD version of the same algorithm has no branching. Each set of  $S$  records is first tested against condition, then processed.

**Sequential:**

```

process1(y) {
  result = y;
  return;
}

```

**SIMD:**

```

SIMD_Process(mask[1..S], y[1..S]) {
  V = SIMD_bit_vector(mask);
  /* V = number between 0 and 2^S-1 */
  if (V != 0) {
    for j = 1 to S
      if ((V >> (S-j)) & 1) /*jth bit*/
        {
          result = y[j];
          return;
        }
  }
}

```

Fig. 4. Return first match algorithm.

Fig. 4 describes the algorithms for returning the first item matching the condition. In database systems this would be for example SELECT \* FROM table WHERE primary key = id.

The sequential version of select first algorithm is straight forward, we return the first record matching the condition. In the SIMD version of the algorithm the mask produced by the SIMD condition operation is transformed into a bit field, where each bit represents one processed record. These are then checked in two layers. First layer, purely an optimization layer, checks if any of the records processed matched, and if so the second layer finds out which one was the first to match.

This example shows that by using SIMD operations in this type of algorithm, the amount of branching has been decreased. In order to estimate by how much, let

$n$  = number of elements to process

$s$  = width of SIMD registers

$m$  = index of the searched element in  $1..n$

$b$  = amount of branches

(seq.)  $b = m$

(SIMD)  $b = \lfloor \frac{m}{s} \rfloor + m \bmod s$

We can see that as the sequential version processes first the condition the total amount of branches is clearly  $m$ . The SIMD versions branch count depends entirely on the magnitude of  $s$ . Thus clearly: if  $s = 1$ , then  $b = m$  and if  $s > 1$ ,  $b < m$ .

By using the masks, one is able to cut the amount of branching instructions, and thus the potential branch mispredictions. If the query searches for the single possible match against the primary key, then there are no differences in the mispredictions for either algorithms. Sequential versions CPU is able to guess the branch as it does not change from the previous run; SIMD would be faster only if the register fits multiple records. But if the condition is more complex, for example  $y < x + 2$ , we can notice how the branching is not related to the previous records at all.

Zhou and Ross noticed that the speed gain from using the CPU SIMD instructions was super linear in some cases. This

### Sequential:

```
process1(y) {
    result = result + y;
}
```

### SIMD:

```
SIMD_Process(mask[1..S],y[1..S]) {
    temp[1..S] = SIMD_AND(Mask[1..S], y[1..S]);
    sum[1..S] = SIMD_+(sum[1..S], temp[1..S]);
}
```

Fig. 5. Cumulative addition algorithm

means that their result was more than linearly better than the sequential version, or better than time of the sequential algorithm divided by the amount of records fitting into SIMD register (amount of parallelism). Fig. 5 shows one of their super linear algorithms.

In the algorithm from fig. 5 the SIMD algorithm transforms the data into 0's for the non matching records. This way they do not affect the addition aggregation, and the algorithm has zero branches. The sequential algorithms branch count is  $n$ . If the query has a complex condition, we can see how the branching sequential algorithm is at a disadvantage when compared to the SIMD version.

Zhou and Ross showed that this method can be applied to a wide range of SQL operations, and that adapting a simple database system to use SIMD is straight forward. They received speedups in nearly all their test cases, and showed converging results for the types of queries that select everything. The reason for this convergence is that as the selection matches everything CPU does no longer mispredict branching.

The paper describes a few advanced topics beyond data selection, namely searching with an index using CPU vector processing. Searching from an index structure with SIMD optimizations was shown to be faster than without, in some cases where branches were difficult to predict the speedup was factor of 9.

### B. Accelerating SQL Database Operations on a GPU with CUDA

In this paper Bakkum and Skadron [4] accelerated SELECT queries with NVIDIA's CUDA framework. They implemented the virtual machine running inside the SQLite database system as a GPU kernel. SQLite uses its own intermediate language, to which SQL queries are compiled. The operations in this language are composited of an opcode and multiple parameters. By implementing these operations in the GPU, they were able to test the benefits of GPGPU against a similar sequential system.

Paper describes the test setup as fair. The sequential test has data loaded into memory, eliminating disk accessing times. SQLite has been compiled with common optimization parameters, making use of some processor specific optimizations afforded by the compiler (this does not make use of the previous SIMD instructions).

The paper admits that SQLite is at a disadvantage, as it does not implement any parallel behavior on the CPU side. This

is similar situation as in Zhou and Ross' paper. Bakkum and Skadron estimate the upper bound for CPU side advancements would be less than the number of threads (assuming there are that many CPU cores), making the speedup maximally linear. Zhou and Ross had super linear results in the CPU side, so this raises concern over the Bakkum and Skadrons papers claims.

Let us read it as an example on how much can a non parallelized non trivial application gain from GPGPU parallelism. The test is also run with special equipment; using high end GPU equipment with large number of cores.

The test discovered multiple problems, that are due to the highly specialized nature of GPUs. They report that not all primitive data types are supported, such as 32 bit integers. Similarly not all GPUs support branching, or longer datatype such as 64 bit floating point values [4].

Despite the problems with the test, the results show speedups of over 20 times. These speedups may be in turn a product of highly restricted environment, they transformed the data from SQLite's B-tree into a row-column format, with fixed column sizes. Also the aggregate functions of SQL (SUM, COUNT, MIN, MAX, AVG) are only supported for the integers.

### C. Comparison

It is difficult to compare these two research papers, mainly as the queries used are different, data sizes and types are not matching, and they are run under different environments by computers of different eras.

In order to provide some figure that may be compared at least in terms, a select query that matches everything in the table (SELECT SUM(X) FROM table) has been used in both test scenarios. Zhou and Ross tested the sequential operation to take about 20 milliseconds, while their SIMD version was around 4 milliseconds. This would mean a speedup of 5x, while Bakkum and Skadron report a speedup around 45x for a similar query. This is not a fair estimate to compare the approaches, since they did not operate on the similarly sized datasets (Zhou and Ross used 1 000 000 rows while Bakkum and Skadron used 5 000 000). As Bakkum and Skadron have not publicized their timings, we can not do a fair comparison.

The papers also state the machine specifications the programs were run on; the Zhou and Ross paper used Pentium 4 type of CPU, with 128 bit wide SIMD registers where Bakkum and Skadron paper used a high end NVIDIA Tesla C1060, with 240 parallel cores. The Pentium 4 is able to use  $\frac{32}{128} = 4$  parallel calculations, and Tesla has 240 cores calculating at the same time. This shows us that the machinery the researchers used differs by a wide margin.

Other than speed, we can compare the amount of work reported. Zhou and Ross describe their effort as "relatively easy", while Bakkum and Skadron talk about "reimplementing the SQLite virtual machine as CUDA kernel". CPU instructions can be used in partial implementations, while GPU effort requires one to go all in.

## IV. CONCLUSION

Database parallelism is a fertile ground for further study. Both papers describe new avenues in which their work can be improved and advanced.

The CPU advancements promised by the Larrabee family of CPU's would have expanded the SIMD registers into 512 bits, allowing more parallelism than the 128 bit registers used in the experiment.

GPU's show ability to calculate large quantities of data efficiently. Main problem with the GPGPU is wide variety of devices and technologies the program must adapt to. When features such as branching, or 64 bit floating points are missing, it may become difficult to transform the real life application into a GPGPU application.

Bakkum and Skadron showed promise in their figures. Even with the largest foreseen SIMD register CPU achieves 4 times the capacity the tested CPU. This figure is significantly less than the similar advancements in the GPU arena.

#### REFERENCES

- [1] A. Ailamaki, D. J. DeWitt, M. D. Hill, and D. A. Wood. *DBMSs on a modern processor: Where does time go?* In Proceedings of VLDB conference, 1999.
- [2] M.J. Flynn, *Very high-speed computing systems* Proceedings of the IEEE, Vol. 54, Issue 12, pages 1901 - 1909
- [3] Peter Boncz, Marcin Zukowski, Niels Nes, *MonetDB/X100: Hyper-Pipelining Query Execution* Proceedings of the 2005 CIDR Conference.
- [4] P. Bakkum and K. Skadron, *Accelerating SQL Database Operations on a GPU with CUDA* GPGPU-3 March 14, 2010.
- [5] J. Zhou, K. Ross, *Implementing Database Operations Using SIMD Instructions* ACM SIGMOD 2002 June 4-6.
- [6] A. di Blas and T. Kaldewey. *Data monster: Why graphics processors will transform database processing* IEEE Spectrum, September 2009.
- [7] S. Ding, J. He, H. Yan, and T. Suel. *Using graphics processors for high performance IR query processing* In WWW 09: Proceedings of the 18th international conference on World wide web, pages 421430, New York, NY, USA, 2009. ACM.
- [8] P. Boncz, S. Manegold, and M. Kersten. *Database architecture optimized for the new bottleneck: Memory access* In Proceedings of VLDB Conference, 1999.
- [9] Intel Inc. *Intel architecture software developers manual* 2001.
- [10] Intel Inc. *Intel C++ compiler users manual* 2001.

**Kalle Kärkkäinen** is a student in the Helsinki University, Computer Science department. He has written his masters thesis on minimizing memory consumption by heuristic task prioritization.