Synthesis of Algorithms for Memory Hierarchies

Andréj Spielmann
I&C, EPFL

Abstract—Every memory or storage component has distinct characteristics such as the speed of reading and writing, necessity of accessing the data in a block-wise fashion or different speeds of random and sequential access. An efficient program must take into account these characteristics as well as the overall structure of the memory hierarchy. Today, these optimizations are usually done by hand in a case-by-case manner.

In this research proposal we look into the matter of modeling a memory hierarchy and some of the techniques that programs apply to use the provided memory and compute resources efficiently. We propose as a research topic to investigate methods how such behaviour could be synthesized automatically.

Index Terms—memory hierarchies, software synthesis, memory optimization

I. INTRODUCTION

In individual storage and memory components in today’s computers and other data processing machines have a broad spectrum of characteristics and details of usage, such as the speed of reading and writing, necessity of a block-wise access to data, difference in the speeds of random and sequential access to data or various caching policies. An efficient use of the memory hierarchy requires programs to take these characteristics into account. Moreover, the issue is made more pressing by the fact that the speed of memory and storage devices is improving slower than its capacity and slower than the performance of CPUs. Thus, in applications were performance is vital, great effort is expended to optimize programs in this respect.

This requires special data structures, data access patterns and coding style. Doing it by hand is tedious and complicated but unfortunately it is still the usual practice. However, one can see recurring patterns in the techniques applied. Therefore there is space for systematizing this process and deriving efficient program implementations automatically.

This is an interdisciplinary topic, which is the reason for an interesting variety in the choice of papers: The first paper, by Aggarwal et al. [1], defines a simple model of a hierarchical memory and derives lower and upper bounds for the running time of several standard algorithms on this model. The second paper, by C. Kim et al. [2], presents a database search tree that stores the search keys in a fashion taking advantage of the structure of the underlying memory hierarchy. The third paper, by Solar-Lezama et al. [3], is a modern example from the field of software synthesis. It offers a deep insight into analyzing code for finding most recent assignments to memory locations.

II. SURVEY OF THE SELECTED PAPERS

A. A model for hierarchical memory

The paper [1], by Aggarwal et al., is nowadays a classical paper that defines the following simple model of a hierarchical memory: The Hierarchical Memory Model (HMM) represents a random access machine where each access to memory location $x$ requires $\lceil \log_2 x \rceil$ time.

Formally, there are an unlimited number of registers $R_1, R_2, R_3, \ldots$ each of which can store an element of some domain specified by the problem. The operations are the same as those of a random access machine [15], except that accessing register $R_i$ takes time $\lceil \log i \rceil$ and executing any operation takes unit time. We allow for $n$-ary operations: the time to perform $R_i \leftarrow f(R_{i_1}, R_{i_2}, \ldots)$ takes time $1 + \lceil \log i \rceil + \lceil \log i_1 \rceil + \lceil \log i_2 \rceil + \ldots$.

This model can be seen as a hierarchy of memory levels, whose size grows exponentially and access cost grows linearly. It can be viewed as an approximation of a computer with several levels of cache memory, a main memory and a hard drive, where each level is slower but several times larger than the previous one. However, it is arguable whether the linear slow-down is realistic. A justification for the $\log x$ cost function is provided in [4]. Later we will also discuss other cost functions, and many of the methods presented for establishing lower and upper bounds on the time complexity of algorithms also apply to them.
The main contribution of this paper is the investigation of lower and upper bounds of the cost of performing matrix multiplication, Fast Fourier Transform, sorting and several other examples on this computational model, and an elaboration of general methods for establishing such bounds.

Polynomially bounded problems that can be solved in time $T(n)$ on a random access machine can clearly be solved in time $O(T(n) \log n)$ on HMM. In some cases, we can do better by cleverly moving pieces of data to a faster level of memory and reusing them several times in this faster location. We call such problems local. On the other hand, for some problems it is possible to prove that the lower bound on their time complexity is asymptotically harder than the respective $T(n)$. These are called non-local and it turns out that computing FFT and sorting are examples of such problems.

Assuming that $\Omega(n^3)$ is a lower bound for matrix multiplication over semi-rings, we will show that this problem is local. However, for several special classes of semi-rings, the broadest being semi-rings additively generated by a finite number of elements, there exist algorithms for matrix multiplication with lower complexity than $\Omega(n^3)$ [14].

To establish an upper bound on the time complexity of a problem, it is enough to exhibit an algorithm with the given running time that solves the problem. The paper provides an $O(n^3)$ algorithm for matrix multiplication and $O(n \log n \log \log n)$ algorithms for FFT and sorting.

More interesting, however, is the method used to obtain lower bounds. We start by defining the threshold cost function $U_m$:

$$U_m(i) = \begin{cases} 0 & \text{if } i \leq m, \\ 1 & \text{if } i > m. \end{cases}$$

In general, if we assume a hierarchical memory model with cost function $f : \mathbb{N} \rightarrow \mathbb{N}$, we mean that accessing the register $R_i$ takes time $f(i)$. The above cost function $U_m$ is meant to be interpreted as counting the number of I/O operations with respect to an external storage on a machine with a main memory of size $m$. We will write $T_f$ for the cost of a computation on a HMM with cost function $f$. We will write $T_m$ for $T_{U_m}$.

**Theorem 2.1:** 1) The I/O complexity of sorting and computing an FFT graph is

$$T_m(n) = \Omega \left( \frac{n \log n}{\log m} - m \right)$$

2) The I/O complexity of the naive algorithm for the multiplication of two $n \times n$ matrices using semi-ring operations is

$$T_m(n) = \Omega \left( \frac{n^3}{\sqrt{m}} - m \right)$$

3) The I/O complexity of searching is

$$T_m(n) = \Omega(\log n - \log m)$$

Proofs for 1.) and 2.) can be found in [6] and [7]. Number 3.) follows by a simple argument on binary decision trees.

The crucial observation now is that any non-negative, monotone non-decreasing cost function with $f(0) = 0$ can be represented as a weighted sum of threshold functions with non-negative weights: Let $\Delta f(x) = f(x+1) - f(x)$. Then

$$f(x) = \sum_{m < x} \Delta f(m) = \sum_{m=0}^{\infty} \Delta f(m) U_m(x).$$

Since the infinite sum has only finitely many non-zero elements, it follows that

$$T_f = \sum_{m=0}^{\infty} \Delta f(m) T_m.$$

The following result follows from this summation formula and Theorem 2.1 by straightforward arithmetic transformations:

**Theorem 2.2:** Let $T$ denote the cost in the HMM model with cost function $f(x) = \lfloor \log x \rfloor$. Then

1) The cost of searching is

$$T(n) = \Omega(\log^2 n)$$

2) The cost of sorting or computing an FFT graph is

$$T(n) = \Omega(n \log n \log \log n)$$

Since on a random access machine the cost of searching is $\Omega(\log n)$ and the cost of FFT and sorting is $\Omega(n \log n)$, it follows that all of these problems are non-local. The same approach can of course be applied to obtain lower bounds for arbitrary non-negative, monotone non-decreasing cost functions such that $f(0) = 0$.

An interesting further point that the paper discusses is that the model makes the assumption that the user explicitly controls the location of data. According to the authors, this was an assumption that is often not the case on real machines and that in fact the user programs in terms of a virtual memory space and it is the operating system that deciding allocates page frames in the physical memory and the hardware that allocates lines in cache. Notice, that it is also an assumption in our project that we are able to control the layout of data in the memory, and thus we will need to override the memory controller and paging policy of the operating system. However, it is a fact that the user has no influence on the caching policy of the CPU.

Memory management algorithms have been extensively studied for two level memory hierarchies. Sleator and Tarjan [13] have proven that in such setting the Least Recently Used (LRU) policy is in a sense optimal: For any sequence of memory accesses and any memory management algorithm A, the cost of the sequence of memory accesses when LRU is used is at most a constant multiple of the cost when policy A is used, provided that the size of the higher level of memory (cache) available to LRU is larger than the cache available to A. The constant is then proportional to the ratio of the cache sizes.

Paper [1] also shows a straightforward generalization of this result to memory hierarchies of more than 2 levels.

A shortcoming of this model is that it does not address some of the more complex properties that can be important in real
world memory hierarchies - blocked access to data, different speeds of reading and writing, advantages of sequential access over random access to data. These issues can not be modeled even with more complex cost functions. A slightly more complicated memory model is presented in [16] and its application to multi core systems in [17].

B. Fast Architecture Sensitive Tree Search on Modern CPUs and GPUs

We discuss paper [2], by Kim et al., because it describes a state-of-the-art example of a data structure that can be optimized for the hardware it runs on by tuning a set of parameters, and it demonstrates many of the techniques used in the field.

Database search using index search trees is a task that is hard to optimize on modern hardware architectures. It involves accesses to memory locations that are hard to predict. In the event of a last-level cache miss or a translation look aside buffer miss, the processor stalls for several hundred cycles. The latency of memory instructions can be reduced by prefetches. These are, however, hard to implement because of the irregular memory access pattern of a tree traversal.

The standard B+-trees [8] were designed to accommodate searches on disk-based systems. As main memories and cache memories have become larger in size, their speed and latencies are now more important for the performance of the search tree than that of the hard disk. Several improvements on B-tree indexes have been proposed to improve cache performance but all have very variable performance depending on the application. A summary can be found in [9].

Several recent system features and techniques can be employed to improve the performance of an index search tree:

- Cache line size and memory page size aware structure of the tree
- Data Level Parallelism - Utilisation of vector processing techniques, usually referred to as SIMD (Single Instruction Multiple Data)
- Thread Level Parallelism - Utilisation of Multiple cores
- Compression techniques

The paper advocates building binary trees as the index structure, with a layout optimized for the specific architectural features. The tree layout presented is called FAST - Fast Architecture Sensitive Tree. Assuming a fixed key size, the nodes in a FAST are blocked in a hierarchical fashion of three levels:

The smallest kind of block consists of the maximum number of keys that can form a complete binary tree and fit together into a cache line. We store them within the second-level block in a breadth-first fashion with respect to their tree arrangement. In a similar fashion as do the first-level blocks, second-level blocks also form a tree.

Thus a third-level block consists of the maximum number of second-level blocks that can form a complete binary tree and can fit together into a memory page, and these are stored in a breadth-first fashion with respect to their tree arrangement. Finally, the third-level blocks are also stored in a breadth-first fashion according to their respective tree arrangement. The key size used throughout the paper is 4 bytes.

For a graphical representation of the level hierarchy of the tree, see Figure 1.

![Level hierarchy of FAST](image)

The authors of the paper claim, unfortunately without a further explanation, that this hierarchical layout provably minimizes the number of accessed cache lines and memory pages. However, the latency when cache line boundaries and memory page boundaries are crossed is still present. Hence to make the best use of the available computational power, it is necessary to have several queries running at the same time.

A large part of the paper is concerned with the particulars of the implementation and analysis of performance of FAST on the Intel Core i7 CPU and NVIDIA GTX 280 GPU. The main focus of the optimizations is on utilizing SIMD and multi core parallelism.

On Intel processors the instructions from the Streaming SIMD Extensions (SSE) instruction set can operate simultaneously on four 32 bit elements, i.e., 4 keys of the assumed size of 4 bytes. Thus the first-level blocking size on the Intel CPUs is 3, which means that 2 levels of the tree are resolved simultaneously in the search operation, providing for a theoretical speedup of 2x in comparison with scalar processing. However, an increased instruction overhead reduces the speedup slightly. On the other hand, the building of the tree can be parallelized by dividing the data equally amongst the available cores.

As mentioned above, the latency of cache misses during a search query can be hidden by processing multiple queries simultaneously. The authors claim that experimentally they found that an optimal number of simultaneous queries per thread is 8 and hence, since a Core i7 supports 8 threads,
to achieve peak throughput we need 64 concurrent queries.

The GTX 280 GPU has 30 Scalar Multiprocessors (SM). SIMD functionality is implemented by each of the SMs having 8 scalar processors that execute the same instruction in parallel. However, every GPU instruction works on 32 data elements and is executed in 4 cycles, so the SIMD width should be viewed as 32. Thus we can take the first-level blocking size to be 15 (this corresponds to a binary tree with 4 levels) and use SIMD to execute two independent queries at the same time. Notice that this is a different strategy than we took with the CPU, where we preferred to execute only one query and instead use SIMD to maximize the first-level blocking size. This is because the CPU does not have support for the so called "warp" operations, which means loading and storing data for a SIMD operation from/to different memory locations, whereas the GPU does.

Since the memory bandwidth of the GPU is less of a bottleneck than on the CPU, it turns out that we only need to double the number of queries per thread to hide the memory latency (in comparison with the factor of 8 needed on the CPU). Since each of the SMs is capable of running 8 threads (called warps in the GPU terminology), we need altogether 30x8x2x2=960 concurrent queries to achieve full load on the GPU.

The performance comparison itself is not of much theoretical interest. An interesting observation, though, is that memory latencies are a bottleneck for the CPU and if the search tree does not fit inside the last level cache, the performance becomes "memory bound" and it can not take advantage of all the available computational power, in particular the SIMD operations. In the end, with all the optimizations mentioned above, the given GPU performs 1.7x faster than the CPU. However, the theoretical memory bandwidth and performance in terms of GFlops of the GPU are 4.7x and 3x larger than that of the CPU. Therefore we can conclude that the CPU search is in fact much better than the GPU search in terms of architecture efficiency.

Since for large trees the search performance on CPUs is memory bandwidth bound, it can be useful to consider applying some kind of compression to the keys. This will trade some computational power (of which we assume there is a surplus) for increasing the memory throughput in terms of the number of keys. There are two scenarios to consider: First, if the keys are of variable length, we need to map them to (shorter) keys of a fixed length, for the blocking schema to be applicable. On the other hand, even if we have keys of a fixed length of \( k \) bytes but they do not take up the whole range, but rather their number is between \( 256^{n-1} \) and \( 256^n \) for some \( n < k \), it is theoretically possible to map them to keys of length \( n \) bytes. In any case it is important that the compression preserves the order of the keys, for range queries to be possible.

The compression schema is presumably not going to be injective, at least in the case of compressing variable-length keys. We build the search tree using the original keys but store the compressed keys. When performing a search, we compress the key in the query, traverse the tree comparing compressed keys and decompress only in the leaf. Since the compression is non-injective, this can lead to landing in an incorrect leaf and having to redo all the work taking a different possible path. Therefore we aim to minimize the number of false positives. This is especially important in the upper levels of the search tree. The authors propose using longer compressed keys (i.e. smaller "compression rate") for the first page of the tree than for the rest.

The speedup achieved by using this compression technique was heavily dependent on the original size of the input keys - for 16 byte keys the speedup was about 1.5x but for 128 byte keys it was already 6x.

We see that the hardware architecture determines many properties of a particular implementation of FAST. One has to decide constants such as block sizes and the number of queries per thread, but also decide about enabling or disabling key compression and choosing the right way of exploiting SIMD operations. All of these decisions can either be directly inferred from specifications of the memory hierarchy (block sizes, SIMD strategy) or be based upon performance testing (number of queries per thread, compression level), and thus they could be the object of an automated analysis.

C. Sketching Stencils

Sketching is a software synthesis approach where the user develops a partial implementation of the program - a function, leaving "holes" in places where the code is hard to write (e.g. loop bounds, array indices, complex arithmetical expressions), and provides a complete specification program written in the same language, presumably using a different and less efficient approach to the same problem that is easier to write. The synthesizer then fills in the holes so that the final program behaves like the specification.

Paper [3], by Solar-Lezama et al., is an extension of previous work by a subset of the same authors, which defined the SKETCH language [10]. The original system works only for small finite programs with bounded inputs. The system developed in this paper works for stencil computations, a large class of programs with unbounded inputs and outputs. Stencil kernels transform an arbitrarily large input grid (i.e. a multidimensional array of elements) into an output grid. Every element of the output grid is obtained by a simple constant-time computation on a bounded number of elements of the input grid, typically from a small neighbourhood of the output element.

First we will briefly describe how the original SKETCH compiler works. As input, the user writes a complete procedural implementation of the program and sketches an outline of a more efficient implementation. The outline will contain "holes" represented by the operator ??, that instructs the compiler/synthesizer that a constant expression needs to be filled in that place. The compiler then pre-processes the code of both the specification and the sketch by unrolling loops an appropriate number of times and inlining all function calls. Every ?? operator is then replaced with a distinct fresh variable, referred to as a control variable. When this is done, the specification and the sketch are translated into Boolean
functions $P(x)$ and $S(x, c)$ respectively, where $x$ represents the program inputs and $c$ the program variables regarded in their bitwise representation.

The sketch resolution can now be stated as a 2QBF Boolean satisfiability problem: $\exists c. \forall x. P(x) = S(x, c)$. This formula is then given to a solver that finds appropriate values for $c$. Notice that this whole translation is made possible by the fact that $x$'s come from a finite domain.

Notice also that, since the functions calls are inlined before the ?? operators are replaced by fresh variables, the semantics of function calls is not completely standard because the ?? operators inside a function can eventually be replaced by different constant values in each use of the function.

The key to applying sketching to stencil computations is that although a stencil works with grids of arbitrary size, every element of the output grid is computed using only a fixed finite number of elements of the input grid, so in fact a stencil computation can be converted into a finite program compatible with the original SKETCH compiler. I will now briefly describe the algorithm:

The input is a pair of programs:

```plaintext
float[N] spec(float[N] in)
{
    ...
    return out;
}

float[N] sketch(float[N] in)
implements spec
{
    ...
    return out;
}
```

Here `spec` is a complete executable program that takes a grid as input an computes an output grid, and `sketch` is a sketch of another program designed to perform the same computation and containing occurrences of the ?? operator. Notice that we assume that the stencil is operating on a one dimensional array. This is just for the simplicity of expressions. All of the following arguments also apply to multidimensional stencils. Suppose now that we are able to obtain code for reduced functions that compute a single element of the output grid:

```plaintext
float reduced_spec(float[N] in, int idx)
{
    ...
}

float reduced_sketch(float[N] in, int idx)
{
    ...
}
```

These functions ought to be such that for any index within the array bounds `spec(in)[idx]` is equal to `reduced_spec(in, idx)` and similarly for the sketched functions. If this holds, then the above programs `spec` and `sketch` are equivalent to the following two programs respectively:

```plaintext
float[N] spec2(float[N] in)
{
    float[N] out;
    foreach j <- [0, N-1]
        out[j] = reduced_spec(in, j)
    return out;
}

float[N] sketch2(float[N] in)
implements spec2
{
    float[N] out;
    foreach j <- [0, N-1]
        out[j] = reduced_sketch(in, j)
    return out;
}
```

Notice that these two programs only differ in the reduced function that they call. Assuming that we have the code for `reduced_spec` and we have a sketched code for `reduced_sketch` containing ?? operators, the problem now reduces to completing the sketch of `reduced_sketch` so that it implements `reduced_spec`. Recall that by our definition of stencil computations, we assume that the reduced functions only in fact operate on a fixed bounded number of elements of the unbounded array that they take as input. To be able to use the original SKETCH compiler to complete the sketch, we would like to transform the reduced functions so that they only take this bounded number of elements as input, instead of the array of unknown size. This can be done using a technique where the arrays are treated as uninterpreted functions, originally proposed by Ackermann [11] and used extensively in hardware verification [12].

We now describe how to obtain the reduced functions, which is the most important contribution of the paper. Both `spec` and `sketch` are reduced using the same algorithm, so I will be describing it in terms of a generic function `float[N] f(float[N] in)` being reduced to `float reduced_f(float[N] in, int idx)`.

We will first define the notion of an execution point. The set of execution points is called $P$ and a $p \in P$ is a pair $(s, t)$ where $s$ is a static instance of a program statement (i.e. one of the commands of the program) and $t$ is a valuation of the loop variables that are in scope at the statement $s$. We are referring to execution points within the original function $f$.

In the construction of the reduced function, we will need the following function:

$$RD : M \times P \times I \rightarrow P$$

$RD$ maps a memory location $m \in M$ (i.e. a variable name or an array name along with an index), an execution point $p \in P$ and a program input $i \in I$ to the most recent execution point prior to $p$ at which an assignment to $m$ occurred. The program input $i$ includes the input grid together with any scalar arguments. In the paper, the most recent assignments are called reaching definitions.
For each scalar variable \( v \) of type \( T \) occurring in \( f \) we define a function
\[
v_{\text{fn}} : P \times I \rightarrow T
\]
that computes the value of \( v \) at a given execution point under a given input, and for each array variable \( a \) of type \( T \) we define a function
\[
a_{\text{fn}} : \text{Int} \times P \times I \rightarrow T
\]
with an analogous effect. We will call these functions \( v \)-functions, where the ‘\( v \)’ stands for “variable”.

Let \( P_e \) and \( P_b \) denote the end and beginning execution points of the program respectively. Then the reduced function can be written as follows:

```c
float reduced_f(float[N] in, int idx)
{
    return out_fn(idx, P_e, in);
}
```

A \( v \)-function for a variable \( v \) (or an array access \( a[idx] \)) at execution point \( p \) under input \( i \) can be constructed by the following algorithm based on syntactic transformation of the original program:

1. Set \((s, t) = RD(v, p, i)\) (or \( RD(a[idx], p, i) \)) and as \( s \) is an assignment statement assigning to \( v \) (or \( a[idx] \)), let \( e \) denote the expression on the right-hand side
2. Return a valuation of a transformed expression \( F(e) \) where each variable sub-expression \( v' \) is replaced with \( v_{\text{fn}}(v', i) \) and each array access sub-expression \( a'[e'] \) is replaced with \( a_{\text{fn}}(F(e'), p', i) \).

Notice that each call to a \( v \)-function returns a value, so computing the valuation of the transformed expression in step 2 amounts only to applying arithmetic operations.

It remains to show how the \( RD \) function computes reaching definitions.

First we define an order on the execution points: Suppose \( p_1 = (s_1, t_1) \) and \( p_2 = (s_2, t_2) \). We say \( p_1 <_P p_2 \) whenever \( t_1 \) is lexicographically before \( t_2 \) or if they are equal \( s_1 \) is located before \( s_2 \) in the code. We obtain a lexicographic order on the valuations of loop variables by treating them them as strings of numbers, where each number corresponds to a valuation of one of the loop variables, written in a suitable order respecting the nesting depth of the loops and the position of different loop nests in the code.

Notice that this defines a total order on all possible pairs \((s, t)\). However, because of conditional guards, not all execution points get executed for every program input. Therefore we define the predicate \( q(p, i) \) which holds if \( p \) gets executed under input \( i \). Formally, \( q(p, i) \) is the disjunction of all path constraints for all paths that reach the execution point \( p \). Notice that if \( p_1 <_P p_2 \) and both \( p_1 \) and \( p_2 \) actually execute then \( p_1 \) executes before \( p_2 \).

Now the procedure \( RD \) can be defined as follows:
\[
RD((a, idx), p, i) = \max\{\{P_b\} \cup \{(s, t) \mid (s, t) <_P p \land q((s, t), i) \land s \equiv a[e] = e' \land e_{\text{fn}}((s, t), i) = idx\}\}
\]
This expression can now be evaluated by iterating through all execution points in a descending fashion in the order described above. However, the compiler uses algebraic reasoning on equalities, inequalities and logical connectives to simplify this procedure. When we have an assignment of the form \( a[g(idx)] = e \), the constraint \( g(j) = x \) often suffices to fully define the iteration space in terms of \( idx \). This concludes the description of the compilation of a sketched stencil.

It could seem that this paper is not directly related to our topic of synthesizing efficient algorithms for memory hierarchies. However, we present it because it gives an example of a method for synthesizing efficient code by applying complex transformations to the existing code. One can increase the rate at which data that is stored together is also accessed together (within a short sequence of instructions) by changing the order in which elements in a loop nest are traversed. This can be done by applying techniques similar to the stencil transformations presented in this paper. The technique of transforming a program whose output is a grid to a program that computes just one element of that grid, and in the computation uses only the bounded part of the input which is necessary, can prove useful in these loop nest transformations.

III. Conclusion and Future Research

We propose to build an automatized code-to-code compiler/synthesizer whose input will be a program written in a style assuming running on an idealized random access machine, and the output will be a program optimized for running on particular hardware, the description of whose memory hierarchy will be provided by the user.

Hence, we will need to design a model for memory hierarchies rich enough to capture the important properties of both present and future storage and memory hardware. The model presented and investigated in [1] is very simple but it is able to describe the differences of capacity and speed between individual levels of a memory hierarchy. It is not, however, able to model some of the more complicated particularities of the hardware, such as differing read-, write- and possibly erase-speeds, access to data only by pages or the advantage of sequential access over random access.

Paper [2] is a state of the art example of the kind of memory-hierarchy-aware algorithm that we would like to be the output of the tool we propose to build. The core principle behind the outstanding performance of the search tree presented is its blocked structure mindful of the sizes of the working units of every level of the expected memory hierarchy (SIMD registers, cache line, memory page). The paper further proposes that in some cases, it can also be advantageous to apply a compression schema to the data (search keys, in this case).
These same techniques can be applied to optimise many other data processing tasks, but they already exhibit a certain amount of dependency on the particularities of the target hardware - the block sizes, the mode of SIMD parallelism and the degree of compression (if any) must be chosen.

Another very important class of optimization methods, which were not directly discussed in the three papers I presented, are loop transformations - blocking, fusion and permutations. These transformations are often employed by compilers with the aim of changing the order in which individual data elements data are accessed so that as much as possible of the data present in a line of the processor’s cache is used before it is replaced. [15].

The know-how from this area could be adapted to automatically apply them also to improve data locality on other levels of the memory hierarchy. The method for transforming a program computing a multidimensional array to a program whose output is a single element of that array presented in [3], and also the analysis of most recent assignments to memory locations, can be very significant in this context.

Although it is impossible to create a tool that will for an input program synthesize the optimal implementation for a given memory hierarchy, we have seen that applying even simple ideas can lead to several-fold performance increases. Presently it is the practice to do these optimisations case-by-case for every application. We believe that if this was done systematically, a lot of labour could be saved and even better results achieved.

REFERENCES