Intel Intrinsics for Breadth-first Search

Tobias Edwards
tedwards@kth.se

January 2021

ABSTRACT Graphs are a common representation for many real-world relationships. One task when using graphs is determining distances between vertices. For this, breadth-first search is a well-known algorithm. However, the inherent properties of common data structures for graph representation make it challenging to efficiently utilize multi-thread and SIMD-instructions to improve BFS performance. In this paper, we look at how the BFS-algorithm can be redefined in algebraic terms. A state-of-the-art data structure, SlimSell, combined with SIMD-instructions for vectorizing the BFS-algorithm is implemented and tested. We compare the results of SlimSell with the compressed sparse row format for sparse matrices. We find that SlimSell overall performs better than CRS. When using SlimSell with multi-threaded systems, the best performance is acquired when $\log_2 \sigma_{opt} = \log_2 n - \log_2 num\_threads$, for a graph with n vertices. The parallel efficiency is worse for SlimSell than CRS in our tests, yet this could be a result of not choosing the optimal σ for SlimSell.

1 Introduction

Within scientific computing, data analysis and other fields, many problems can be structured as graphs [1, 2]. One of the common graph problems is determining shortest paths (number of hops) between two vertices. For such a problem, the traditional breadth-first search algorithm (BFS), utilizing a while-loop and a queue, is perhaps one of the most well known and studied graph search algorithms.

However, while the traditional BFS might suffice for smaller problem sizes, as graph sizes grow (both in edges and vertices), the need for data structures and algorithms that efficiently utilize hardware implementations becomes more apparent. For instance, the traditional BFS can introduce irregular memory accesses that do not benefit from spatial or temporal locality [1].

Another challenge with the traditional BFS-algorithm is introducing parallelism to increase performance [**1**]. For high-level parallelism, such as multi-threads, dividing the computations evenly across threads is important for equal-sized workloads. However, in the traditional BFS, expressing evenly-sized subgraphs a priori to computation is a difficult problem [1]. Also, as stated previously, due to the irregular data accesses in the traditional BFS, utilizing data-level parallelism it not feasible.

Computer processors have been extended with single instruction, multiple data (SIMD) instructions to support data-level parallelism in order to more efficiently perform operations across multiple data element [3]. *Vectorization* is a programming model which focuses on operating on arrays of elements instead of individual elements of the array [4]. Such a programming structure allows for better use and optimization of the SIMD-instructions.

In this paper, we look at how the BFS-algorithm can be redefined in terms of linear algebra in order to utilize available SIMD-instructions in the x86_64 AVX2-extension. This *algebraic* BFS-algorithm focuses on performing matrix-vector multiplications over the *tropical semiring*, meaning we (respectively) replace addition and multiplication with the minimum operation and addition, to perform the BFS. In order to more efficiently utilize SIMD-instructions, a data structure, *SlimSell*, is presented by [2].

The goal of this project is to perform our own tests using the SlimSell format for BFS, and then comparing the results with BFS using compressed row storage (CRS) format. We discuss factors that can affect the performance of BFS using SlimSell.

2 Breadth-first search

The purpose of the BFS-algorithm is to find the shortest distances (in number of steps) from a root vertex $r \in V$ in a graph $G = (E, V)$ to all other vertices of V, where E and V are the edges and vertices of the graph, respectively. In this project, G is an undirected and unweighted graph that is not necessarily connected. We will use the notations $n = |V|$ and $m = |E|$ in this paper to refer to the cardinality of edges and vertices.

The defining aspect of BFS is to maintain a *frontier*, the set of vertices in V that have the same distance to the root vertex. For example, the first frontier consists only of the root, which has distance 0, the second frontier consists of all vertices that have an edge to the root, which have distance 1 to r, and so on.

The result of the BFS is a vector $\mathbf{d}$, containing the distance from the root vertex to all other vertices of G. Vertices that are unreachable should have a value in $\mathbf{d}$ to identify this, -1 for instance.

2.1 Traditional BFS

The traditional BFS-algorithm uses a while-loop and a FIFO-queue. Iterating over the current frontier and adding the next frontier vertices that haven't been visited yet.

Algorithm 1 shows how the traditional BFS algorithm can be implemented using vertex identifiers $\{0, ..., n-1\}$ and adjacency list L_G to represent G (each element of L_G contains a vector of vertex identifiers indicating adjacent vertices).

Algorithm 1 Traditional BFS
Input: Adjacency list L_G, root vertex r
Output: Vector $\mathbf{d}$ with $\mathbf{d}[i]$ holding the distance of the shortest path from v_r to $v_i \in V$

```
1: for $i \in \{0, ..., n-1\}$ do
2:     $\mathbf{d}[i] \leftarrow -1$
3: end for
4: $d[r] \leftarrow 0$
5: $Q.push(r)$
6: while not $Q.empty()$ do
7:     $j \leftarrow Q.front()$
8:     $Q.pop()$
9:     for $i \in L_G[j]$ do
10:        if $\mathbf{d}[i] = -1$ then
11:            $\mathbf{d}[i] \leftarrow \mathbf{d}[j] + 1$
12:            $Q.push(i)$
13:        end if
14:    end for
15: end while
```

Algorithm 1 can be modified to parallelize the BFS. This could be done by dividing the current queue (frontier) over threads (line 7), and also when processing each vertex's adjacent vertices (lines 9-13). However, for the parallel algorithm to be correct, we would have to add certain atomic operations on lines 9 and 11 - 13 in order to maintain correctness [5].

2.2 Algebraic BFS

The algebraic BFS-algorithm that we will be using builds on the duality of graphs and matrices, and the tropical semiring, both of which we briefly present here.

2.2.1 Graph - matrix duality

The graph - matrix duality describes the relationship between the common graph representation using collections of vertices and edges, and the matrix representation of said graph [6].

An important property of the matrix representation is that a single step in the traditional BFS-algorithm (searching the current frontier) is equivalent to performing a single matrix-vector multiplication:

$$y = \mathbf{A}^T x \tag{1}$$

where $\mathbf{A}$ is an $n \times n$ adjacency matrix of $G = (E, V)$ such that

$$A_{ij} = 1 \ \forall i, j \in \{0, .., n-1\} \text{ if } (v_i, v_j) \in E$$
$$A_{ij} = 0, \text{ otherwise}$$

and x is a vector of size n such that

$$x_i = 1 \text{ if } i = r$$
$$x_i = 0, \text{ otherwise.}$$

Note that in our case of undirected graphs, $\mathbf{A}$ is symmetric, thus we can ignore the transpose in equation 1.

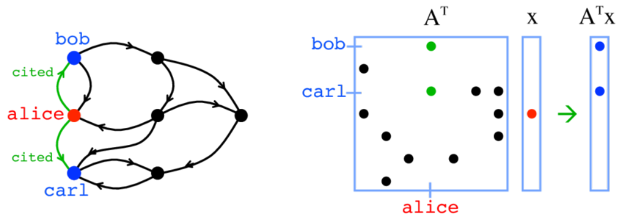


Figure 1: Image from [7]. The example shows a single BFS-step with matrix-vector multiplication. Starting from the red vertex (alice), we can reach the blue vertices (bob and carl) in one step, which is the result of multiplication.

Figure 1 shows an example of a single BFS-step with matrix-vector multiplication.

2.2.2 Tropical semiring

A *semiring* is an algebraic structure that combines a set of elements with two binary operators, often referred to as addition and multiplication [6]. The semiring notation $(S, +, \cdot, 0, 1)$ expresses the set S, with addition and multiplication and corresponding identity elements. Semirings have the following properties for $a, b, c \in S$:

- Both operations of the semiring have identity elements, referred to as 0 (addition) and 1 (multiplication).
- Both operations are associative, e.g. $(a + b) + c = a + (b + c)$
- Addition is commutative, e.g. $a + b = b + a$
- Multiplication distributes over addition from both left and right, e.g. $a \cdot (b + c) = (a \cdot b) + (a \cdot c) = (b + c) \cdot a$
- The additive identity is a multiplicative annihilator, e.g. $0 \cdot a = a \cdot 0 = 0$

Depending on the graph problem, different semirings can be applied [2]. For the BFS-algorithm, we will use the tropical semiring:

$$(\mathbb{R} \cup \{\infty\}, min, +, \infty, 0).$$

Intuitively, the tropical semiring replaces addition with the minimum operator *min*, which returns the minimum value of two operands, while addition replaces multiplication. The identity element corresponding to the *min* operation is ∞, as

$$min(a, \infty) = a, \ a \in \mathbb{R}.$$

As an example, consider the matrix-vector multiplication over the tropical semiring below.

$$\begin{pmatrix} 4 & 8 \\ \infty & -1 \end{pmatrix} \begin{pmatrix} \infty \\ 2 \end{pmatrix} =$$
$$\begin{pmatrix} min(4 + \infty, 8 + 2) \\ min(\infty + \infty, -1 + 2) \end{pmatrix} = \begin{pmatrix} 10 \\ 1 \end{pmatrix}$$

2.2.3 Algebraic BFS-algorithm

Using the tropical semiring we can perform iterations of matrix vector multiplications to compute the distances from the root vertex r. In order to do so we alter the adjacency matrix and right-hand side vector in the matrix-vector multiplication.

Firstly, all non-diagonal zeros of $\mathbf{A}$ are replaced with ∞, which gives us a new matrix $\mathbf{A}'$. As we use the minimum operator to determine current shortest distance, zeros (which indicate no edge) could lead to incorrect distances [2].

Secondly, the right-hand side vector of equation 1 is initialized (for the first iteration only) such that $x_r = 0$ and $x_v = \infty$, $\forall v \in V : v \neq r$ [2].

We will use a sufficiently large integer to represent ∞ in our implemented algebraic BFS-algorithm. As we have n vertices, the longest path possible is one that visits every vertex in G. Thus the path has at most $n - 1$ steps. We therefore choose n as our large number as no real path can be equal or longer than this.

The algorithm for algebraic BFS can be viewed in algorithm 2 below.

Algorithm 2
Algebraic BFS
Input:
$\mathbf{A}'$ (same as $\mathbf{A}$ but off-diagonal zeros are ∞),
root vertex v_r
Output:
Vector $\mathbf{d}$ with $\mathbf{d}[i]$ holding the distance of the shortest path from v_r to $v_i \in V$

```
1: for $i \in \{0, ..., n-1\}$ do
2:     $d[i] \leftarrow \infty$
3:     $x[i] \leftarrow \infty$
4: end for
5: $d[r] \leftarrow 0$
6: while $x \neq d$ do
7:     $x \leftarrow d$
8:     $d \leftarrow \mathbf{A}x$     ▷ mv-multiplication in trop. semi.
9: end while
```

3 Sparse matrix formats for graphs

The graphs used in this project are Kronecker power-law graphs. Such graphs have a high number of vertices with a low degree, while only a small set of the vertices have a high degree. The motivation behind this choice comes from that the scaling of edges in such a graph are considered to represent many real-world relationships, such as number of contacts on social media. Furthermore, these graphs are used by the Graph500 benchmark [8].

The Kronecker graphs are considered sparse in adjacency matrix format, as most vertices of G only have a small number of edges. Thus, the matrix-vector multiplication on line 8 of algorithm 2 is really a sparse matrix-vector multiplication (SpMVM). Optimizing this operation will help us improve the performance of our algebraic BFS.

In general, using an $n \times n$ matrix in memory is often an inefficient representation of a sparse matrix and can slow down the performance of sparse matrix-vector multiplication. This is due to the fact that the number of non-zero elements (NNZ) of the matrix is often considerably smaller than the total number of elements of the matix. As the zero elements of the matrix will not alter the result, avoiding having to store, load and operate on zeros in our system is preferable so as to not waste computation time.

To reduce the computation time of SpMVM we must reduce the storage size of $\mathbf{A}'$. We will first show how our algorithm can be implemented using CRS. However, we will see that even though this format does reduce the the space used for storing the matrix, CRS does not directly allow data-parallel operations. Thus, we turn to SELL-C-σ, another sparse matrix format, that does permit direct use of data-parallelization.

3.1 Compressed row storage

The compressed row storage format (CRS) uses three 1D-arrays: *val, col* and *row* in order to represent a sparse matrix in row-major order. *val* and *col* are arrays of size equal to the number of non-zero elements (NNZ) and contain all non-zero elements of the matrix, and column indices of each non-zero element in the matrix, respectively. *row* is a vector of size $n + 1$ which contains the indices of the corresponding first non-zero element per row in *val* and *col* [9].

Algorithm 3 presents the SpMV multiplication using CRS-format over the tropical semiring. Algorithm 4 uses the same logic as the CRS-algorithm 3, however with four-way modulo unrolling, as presented in [9].

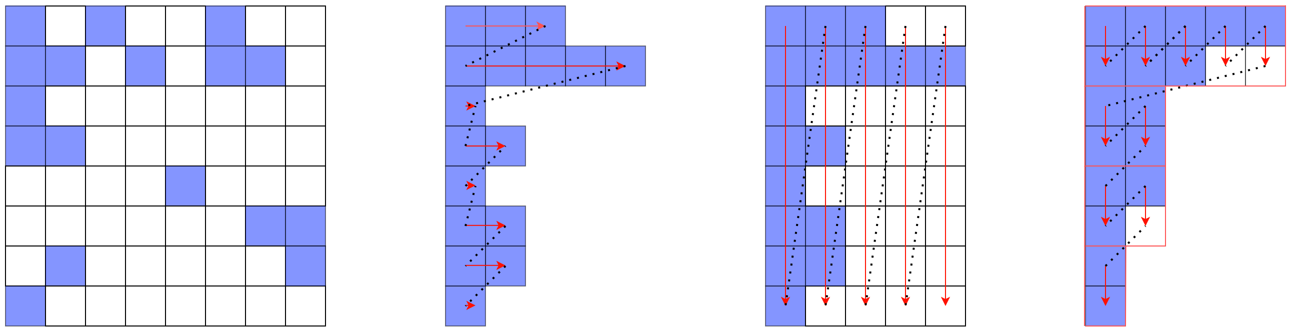


Figure 2: Sparse matrix storage formats. Colored tile indicates non-zero element, clear tiles indicate padding. From left to right: original 8×8 matrix, CRS, ELLPACK and SELL-2-8 (chunk size $C = 2$ and full sorting $\sigma = 8$). Orange arrows indicate how elements are stored in memory. The orange borders on the SELL-C-σ format indicate the four chunks.

Algorithm 3
SpMVM over tropical semiring with CRS
Input:
matrix in CRS-format, dense vectors x and y
Output:
lhs vector y of result of matrix-vector mult.

```
1: for $i \in \{0, ..., n-1\}$ do
2:     for $j \in \{row[i], ..., row[i+1]\}$ do
3:         $y[i] \leftarrow min(\ y[i]\ ,\ val[j] + x[col[j]]\ )$
4:     end for
5: end for
```

Algorithm 4
SpMVM with CRS and 4-way unrolling
Input:
matrix in CRS-format, dense vectors x and y
Output:
lhs vector y of result of matrix-vector mult.

```
1: for $i \in \{0, ..., n-1\}$ do
2:     $tmp_0, tmp_1, tmp_2, tmp_3 \leftarrow 0$
3:     for $j \in \{row[i], ..., row[i+1]-3\}$ do
4:         $tmp_0 \leftarrow tmp_0 \oplus val[j+0] \otimes x[col[j+0]]$
5:         $tmp_1 \leftarrow tmp_1 \oplus val[j+1] \otimes x[col[j+1]]$
6:         $tmp_2 \leftarrow tmp_2 \oplus val[j+2] \otimes x[col[j+2]]$
7:         $tmp_3 \leftarrow tmp_3 \oplus val[j+3] \otimes x[col[j+3]]$
8:     end for
9:     $y[i] \leftarrow tmp_0 \oplus tmp_1 \oplus tmp_2 \oplus tmp_3$
10:    for $j \in \{row[i+1]-3, ..., row[i+1]\}$ do
11:        $y[i] \leftarrow y[i] \oplus val[j] \otimes x[col[j]]$
12:    end for
13: end for
```

Unrolling the inner loop of algorithm 3 by a factor 4 allows vectorizing the algorithm. Instead of performing a single add and multiplication per non-zero element of the sparse matrix, we can utilize SIMD-instructions to perform the add and multiplication over 4 elements simultaneously.

However, an overhead in algorithm 4 is introduced for the reduction and loop remainder on lines 9 and 10 to 12, respectively. These overheads increase with larger SIMD widths. Thus, for the CRS-format we require the number of non-zero elements per row to be significantly larger than the SIMD width to mitigate the impact on performance from the overheads [9]. As a result, CRS is not an optimal choice for high-performance BFS on highly sparse matrices [2].

3.2 Data-parallel formats

In order to overcome the drawbacks of the vectorized SpMVM-CRS, the authors of [9] suggest an alternative sparse matrix format: SELL-C-σ, which is built from the ELLPACK format. In this section, we first present ELLPACK, and then the SELL-C-σ.

Algorithm 5
SpMVM with SELL-4-σ and 4-way unrolling
Input:
matrix in SELL-4-σ-format, dense vectors y and x
Output:
lhs vector y of result of matrix-vector mult.

```
for $i \in \{0, ..., \frac{n}{4}\}$ do
    for $j \in \{0, ..., cl[i]\}$ do
        $y[i*4+0] \leftarrow y[i*4+0] \oplus val[cs[i]+j*4+0]$
        $\otimes\ x[col[cs[i]+j*4+0]]$
        $y[i*4+1] \leftarrow y[i*4+1] \oplus val[cs[i]+j*4+1]$
        $\otimes\ x[col[cs[i]+j*4+1]]$
        $y[i*4+2] \leftarrow y[i*4+2] \oplus val[cs[i]+j*4+2]$
        $\otimes\ x[col[cs[i]+j*4+2]]$
        $y[i*4+3] \leftarrow y[i*4+3] \oplus val[cs[i]+j*4+3]$
        $\otimes\ x[col[cs[i]+j*4+3]]$
    end for
end for
```

3.2.1 ELLPACK

ELLPACK is a spare matrix storage format which stores the sparse matrix in column-major order where all rows (of the sparse matrix) are padded to the length of the row with the most non-zero elements. See figure 2, third format from the left. Using two arrays of equal length, *val* and *col*, we can store the values and columns of each non-zero element in the ELLPACK format. ELLPACK is similar to CRS in that we remove all zero elements, but we now add some padding in order to have equal lengthed rows. ELLPACK permits better parallelization over rows and coalesced memory accesses, thus making it highly application for general-purpose computing on GPUs [9].

Sliced ELLPACK proposed by [10], divides the ELLPACK-format into equal sized slices S. We will instead refer to these slices as *chunks*, where each chunk consists of C rows of the ELLPACK-format. Within each chunk, the rows are padded to the longest row of the corresponding chunk, instead of the longest row globally. The decrease in padding can improve performance as spatial locality is more efficient (memory loads contain less redundant padding).

3.2.2 SELL-C-σ

SELL-C-σ builds on the sliced ELLPACK format with the goal of further reducing the overhead from padding. The idea is that the sliced ELLPACK is most efficient when the padding is minimal. As the amount of padding per chunk is dependant on the length of the longest row per chunk, we can minimize the padding by grouping rows of similar lengths into the same chunks. Thus the SELL-C-σ format is parameterized over C, the "width" of the chunks to divide the rows over, and σ, the sorting degree (number of consecutive rows to sort). No sorting and $C = n$ (SELL-n-1) is equivalent to ELLPACK. If we also set the chunk size C to 1 (SELL-1-1), the storage is identical to CRS. In figure 2, we have $C = 2$ and full sorting $\sigma = 8$ [9]. The sorting degree can range from 1, to the number of rows in the matrix and is normally chosen as a multiple of the chunk size.

To implement SELL-C-σ we, similar to ELLPACK, require 1D-arrays *val* and *col*. Now however, the data is stored in "chunk-column"-order: each chunk of C rows is stored consecutively in *val* and *col* column-by-column. See figures 2 and 3 for clarity. We also require two other arrays *cs* and *cl*, storing the starting offset of each chunk and the length of each chunk, respectively. Thus, cl_i holds the length of the longest row of chunk i, cs_i gives the starting index in *val* and *col* for chunk i [9]. See algorithm 5 for SpMVM with SELL-4-σ and 4-way unrolling. Notice that no reduction or loop remainder is needed, making the implementation highly available for data-parallel operations as the left-hand side and *val* and *col* arrays are stored consecutively in memory.

The SELL-C-σ format, in contrast to ELLPACK, can vary the amount of padding depending on choices of σ and C. This leads to the construction of variable β, which is defined as the "chunk occupancy":

$$\beta = \frac{\text{NNZ}}{\sum_{i=0}^{N_c-1} C \cdot cl[i]} \tag{2}$$

where NNZ is the number of non-zero elements in the original matrix, N_c is the number of chunks that the rows have been divided into and $cl[i]$ is the length of chunk i [9].

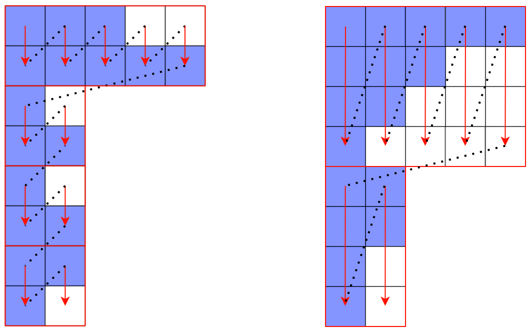


Figure 3: More examples of SELL-C-σ of the matrix in figure 2. The left figure represents SELL-2-1, the right SELL-4-4.

3.3 SlimSell

We have presented the CRS and the SELL-C-σ formats for sparse matrices. We now present an optimization of the formats that are used specifically for graph operations such as BFS.

We can reduce the size of the formats (and thus reduce number of memory accesses) by noticing that, for representing graphs, the *val*-array contains redundant information as it only states whether an edge exists between two vertices. Thus, we can remove this array and simply use the *col*-array. For the SELL-C-σ format, we can use an identifier for padded cells (non-existent edges) such as -1. This reduced format is *SlimSell* [2].

4 Parallelizing BFS

In previous sections, we have described both how to define the BFS in order to vectorize the algorithm, and a data structure, SlimSell, that allows for simple use of vector instructions. In this section, we describe the actual parallelization of the BFS. We describe SIMD in more detail and also present the matrix-vector multiplication using Intel Intrinsics. The Intrinsics are a technology for utilizing vector instructions in C without writing assembly level code [11]. Finally, we also describe OpenMP, an interface for utilizing multiprocessing.

4.1 SIMD

SIMD, single-instruction, multiple-data, is an extension to the processor architecture in order to support data-parallel instructions. In practice, we mean special low-level instructions that perform a single instruction across multiple data. As an example, consider code listing 1, where we perform a linear vector addition.

```
    void add(vec a, vec b, vec c, int n) {
        for(int i = 0; i < n; ++i)
            c[i] = a[i] + b[i];
    }
```

Listing 1: Vector addition

The linear addition in listing 1 works. However, CPUs with *vector* extensions (SIMD-instructions) have special hardware that support operating on multiple data. Consider figure 4. If we vectorize the addition in listing 1, we would modify the code such that rather than iterating over every element, we iterate over consecutive chunks of the vectors and operate on the chunks rather than the elements.

This modification introduces us to SIMD-*width*, meaning the number of elements processed together. Larger widths allow for more elements to be processed together. For instance, if we are using SIMD-width 4 in figure 4, we will require a total of four SIMD loads, two SIMD additions and two SIMD stores to perform the addition. If we instead use SIMD-width 8, we can half the number of SIMD-instructions.

Streaming SIMD Extensions (SSE) is a SIMD implementation for the x86 instruction set architeture (ISA) introduced in the 1990's by Intel. SSE supports 128-bit registers, meaning we can for example operate simultaneously on four 32-bit integers or eight 16-bit integers or two double precision floats. Later, around mid-2010's, Advanced Vector Extensions 2 (AVX2) was deployed which supports many 256-bit instructions. Thus allowing for twice the width of SSE [3].

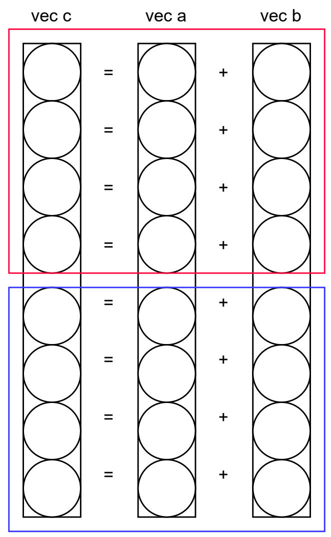


Figure 4: Vector addition visualizing how consecutive elements in the vectors can be grouped and operated on together. The SIMD-width is 4 if we operate on the elements in the red rectangle and the elements in the blue rectangle separately. If we operate over all elements simultaneously, the SIMD-width is 8.

Using SIMD instructions can be an efficient strategy for improving application performance. Tasks such as computer graphics, data mining, image processing and audio and video encoding are a few examples of computationally-intense processes which can benefit from SIMD [3].

4.2 Intel Intrinsics

Intel Intrinsics provide a library to SSE/AVX-instructions that are wrapped in a user-friendly manner for use in C/C++.

The BFS-algorithm we use has 32-bit integer vertex identifiers. As the available hardware for our tests supports up to AVX2, we can at most use a SIMD-width of 8.

The Intrinsic instructions that we use in our code can be found in code listing 2 of appendix A. Finally, the tropical matrix-vector multiplication using SIMD-width 8 can be viewed in listing 3 of the appendix. The function is based on listing 6 from the paper on SlimSell [2].

4.3 OpenMP

The environment that is used for our tests not only supports SIMD, but also has hardware for executing multiple threads of instructions in parallel, i.e. multiprocessing.

OpenMP is an interface for utilizing multiple threads in a *shared-memory* environment. Meaning that multiple threads have access to the variables and data of the current program.

Our use of OpenMP is mainly for dividing the matrix-vector multiplication across threads, as we wish to utilize as much of the available hardware as possible.

We use OpenMP to divide iterations of the for-loops on line 1 in both algorithms 3 and 5 over the available threads.

We can choose the type of thread-scheduling. The default is *static* scheduling, meaning that the for-loop is divided equally across the threads. For example, if we have four threads and a for-loop with 1000 iterations, with static scheduling, each thread will process a consecutive chunk of 250 iterations. If we instead use *dynamic* scheduling, the threads will be given an iterations to process before acquiring the next iteration. This can be useful when the workload in each iteration is not constant. However, dynamic scheduling requires more synchronization than static scheduling, which can add an overhead.

5 Setup and testing environment

For the tests, we compute BFS using the CRS and SlimSell formats. We will occasionally (mainly in section 6) refer to SlimSell as SELL-C-σ in order to be clear on the sorting degree and chunk size. However, remember that SlimSell is basically SELL-C-σ without the *val*-array. The graphs used are Kronecker power-law graphs. Most of the tests are run with graphs with number of vertices $n = 2^{23}$ and the average vertex degree $\bar{\rho} = 16$. We will inform the reader of the graph used for each result. The graph generator is provided by the Graph500 benchmark. The graphs used are all generated with random seeds 2 and 32 [8].

The tests are run on a single node of Beskow, a Cray XC40 system maintained by PDC in Stockholm, Sweden. One node consists of two Intel Xeon E5-2698 v3 Haswell CPUs, running at 2.3 GHz each. Each CPU has 40 MB of Intel Smart Cache and 16 cores, each with 32 threads. A node has 64 GB of main memory and the environment is expected to support Intel AVX2 [12].

We compile with the Intel compiler environment (v 6.0.7) and the following flags:

```
-fopenmp -std=c++11 -g -Wall -xCORE-AVX2
```

The entire code can be viewed at the GitHub repository: `https://github.com/tobzed/project_sci_comp`. For CRS and SlimSell formats, we are able to supply a file containing a graph (generator from Graph500) in row-major order. We can then measure the construction time from file and create statistics for BFS-computation and matrix-vector multiplication.

6 Results

In this section, we present results for a variety of tests with the SlimSell format. We also present results for a BFS-CRS version as reference. The CRS version uses the standard matrix-vector multiplication without loop unolling, as presented in algorithm 3. Both the SlimSell and the CRS formats use OpenMP to parallelize the BFS. All mean values are computed from 128 runs. Standard deviation is presented in some of the results.

For SlimSell, although we can pick any chunk size $0 < C \leq n$, we use a chunk size equivalent to the desired SIMD-width. Therefore, in our tests we mainly use a chunk size of 8, occasionally a chunk size of 4.

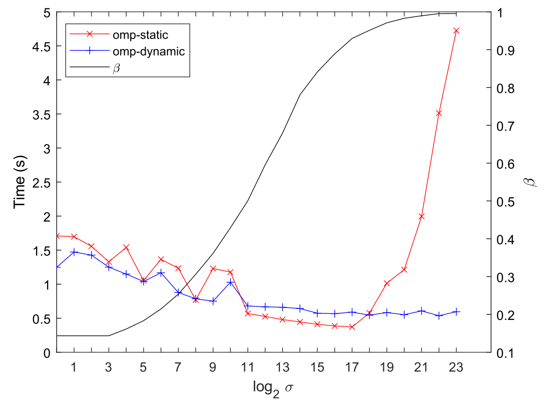


Figure 5: BFS on graph with $n = 2^{23}, \bar{\rho} = 16$ with SELL-8-σ, where $\log_2 \sigma \in \{0, 1, 2, 3...23\}$. Figure shows the mean BFS-time for static and dynamic OpenMP scheduling, and β for the given graph and σ-value.

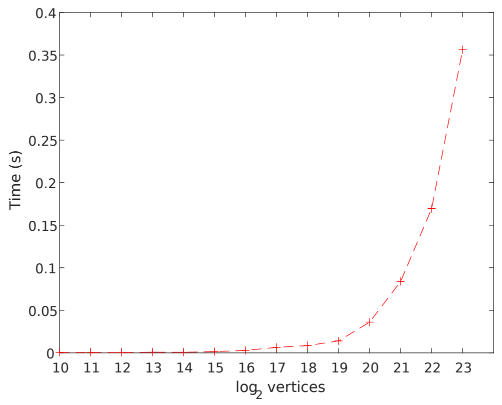
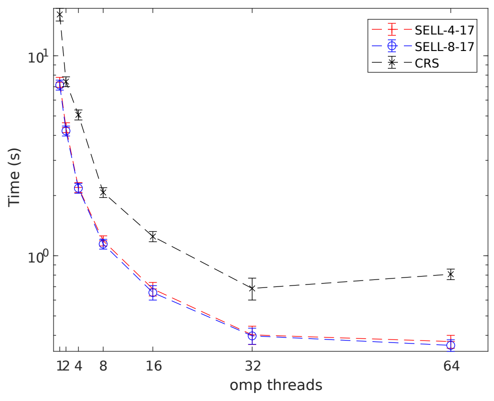
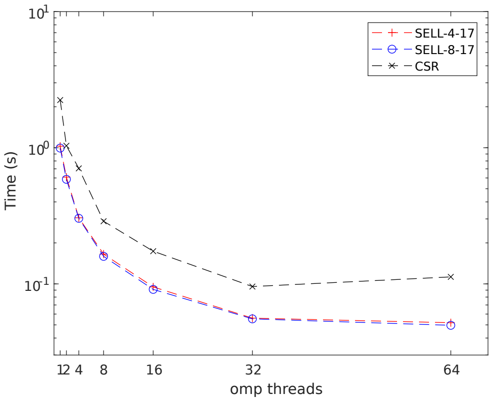


Figure 6: Mean BFS times for SELL-8-σ over increasing scale of graph. We used $64 = 2^6$ OpenMP threads, $\sigma = \log_2(n) - 6$. The average vertex degree was constant ($\bar{\rho} = 16$).

Figure 5 presents the results of BFS-SlimSell for an increasing σ on a graph with 2^{23} vertices and average vertex degree $\bar{\rho} = 16$. The results also show how β changes as the sorting degree increases. The tests

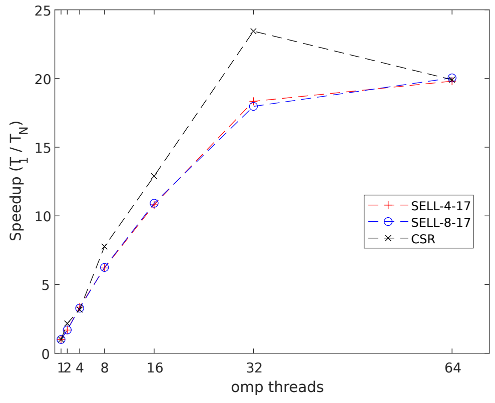


(a) Mean BFS times.

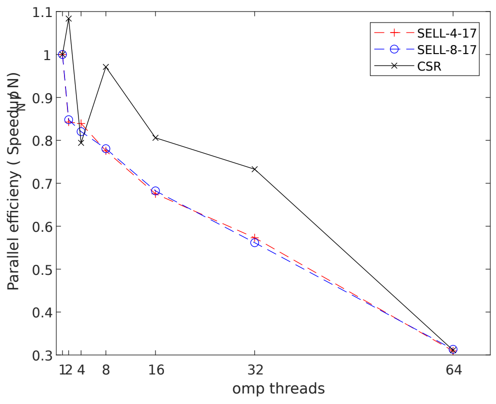


(b) Mean times for matrix-vector multiplication.

Figure 7: Mean times for both BFS and the matrix-vector multiplication operation over the tropical semiring using CRS, SELL-4-17 and SELL-8-17 formats. The x-axis is increasing number of OpenMP threads. Graph had $n = 2^{23}$, $\bar{\rho} = 16$.



(a) Speedup, $\frac{T_1}{T_N}$.



(b) Parallel efficiency, $\frac{Speedup}{N}$.

Figure 8: Speedup and parallel efficiency for tests in figure 7. N = number of OpenMP threads, T_N = BFS time for N threads.

were run using both static and dynamic OpenMP scheduling of the outer for-loop of algorithm 5, where the unrolling has been vectorized (SIMD-width 8).

Figure 6 show how the mean time per BFS, using SELL-8-σ changes with the problem size. We varied the scale of the graph from 2^{10} to 2^{23} vertices but kept the average vertex degree costant: $\bar{\rho} = 16$. The number of OpenMP threads was constant and we used a different σ depending on scale of the graph, see section 7 for more on the choice of σ.

Figure 10 presents the mean BFS time for SELL-8-σ for increasing σ, similar to figure 5, but for other graph sizes and only using static OpenMP scheduling.

Figure 7 presents the results of varying the number of OpenMP threads for the BFS. Figure 7a has the mean BFS time, while figure 7b is the mean time for the matrix-vector multiplication.

Notice that we use two different SlimSell formats:

SELL-4-17 and SELL-8-17. The graph used in the tests had 2^{23} vertices and vertex degree $\bar{\rho} = 16$. We use static OpenMP scheduling.

Figure 8 gives the parallel performance for the testse run in figure 7. We see the speedup and then parallel efficiency for increasing number of OpenMP threads.

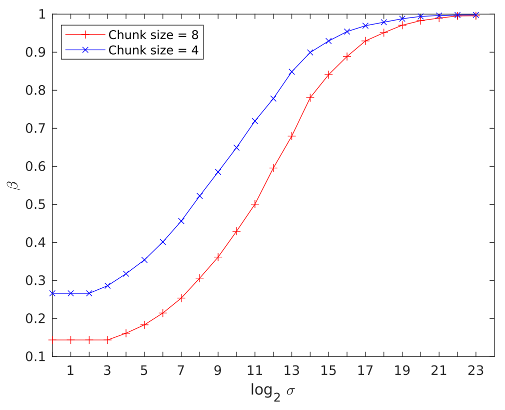


Figure 9: β-values over increasing σs for chunk sizes of 4 and 8. Graph had $n = 2^{23}$ and $\bar{\rho} = 16$.

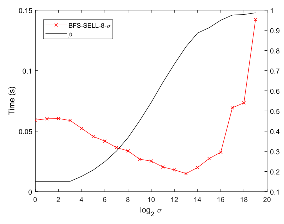
Finally, figure 9 reveals how the β-value changes for different chunk sizes with increasing σs. As we have run tests for SIMD-widths 4 and 8, those are also the chunk sizes thar were used during the tests.

7 Discussion

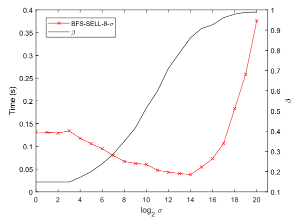
We start off by noting a potential error in the code that was used for computing the results. In the SlimSell format, we use integer -1 to identify padding. In listing 3 of appendix A, lines 32-39, we could potentially perform an out-of-bounds access to the right-hand side vector. This occurs when there is padding in the current column. This results in an undefined behaviour. The error was corrected by loading dummy values for padded values (-1) and then mask them. This fix added a few more instructions to the SlimSell BFS-MVM. The modified code was tested locally and gave similar results as we presented in section 6, with a slightly worse, but still better than CRS, mean time for BFS. We therefore determine that the conclusions we draw will still hold.

Using OpenMP can help improve the performance of the BFS. Figure 7 shows this for both CRS and SlimSell formats. As the number of threads are doubled for each data point, we, at best, hope for a linear improvement in performance (doubling the threads gives almost double the performance). We are not quite there, as we see from 8. Also, notice that the effects of the SIMD-width are barely noticeable, with the mean time for BFS varying between $200 - 20$ ms for SIMD-widths 4 and 8. At first glance, one might expect the SIMD-width of 8 to be twice as fast as the width of 4. Yet this is not the case. We see two main explanations for this discrepancy. Firstly, in figure 9, notice that doubling the chunk size from 4 to 8 has quite a dramatic effect on β. Remember that β indicates the ratio between the original edges of the graph and the total size of the *col*-array of SlimSell. For instance, with $\log_2 \sigma = 11$, the difference in βs for the two chunk sizes is almost 0.23. A larger chunk size means that more rows in the data structure are being padded to the length of the longest row in the current chunk. Thus, smaller chunk sizes mean less padding, which in turns results in less manipulating redundant information. Secondly, using a SIMD-width twice as large, does not necessarily imply twice as efficient instructions. Some of the 256-bit instructions have slightly higher latency than the 128-bit versions. Thus, expecting a doubling of the performance from SIMD-width is incorrect.

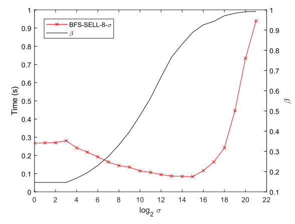
We see from our results (figure 10) that initially increasing σ does improve performance. However, performance is unaffected for $\sigma \leq C$. This is expected as only reordering rows within a chunk will not change β, thus we shouldn't see much change in performance. This is apparent in graphs of figure 10. Notice how the performance is unchanging while $\log_2 \sigma < 3$. However, for the graph with 2^{23} vertices (figure 5), the results are somewhat different for $\log_2 \sigma < 11$, compared to the results in figure 10. The performance is not strictly improving for $3 < \log_2 \sigma < 18$. The reason for this is not entirely clear and may well be an error in the tests. As the only thing that ought to change during this test is β,



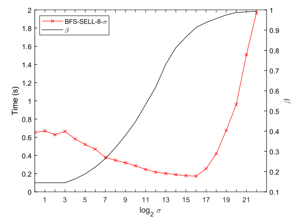
(a) $n = 2^{19}$, $\bar{\rho} = 16$



(b) $n = 2^{20}$, $\bar{\rho} = 16$



(c) $n = 2^{21}$, $\bar{\rho} = 16$



(d) $n = 2^{22}$, $\bar{\rho} = 16$

Figure 10: Mean times of BFS with SELL-8-σ for different sized graphs and σs.

as σ increases. As β is improving (getting closer to 1.0) we should see that the performance also improves (as shown in figures 10).

Notice that, for static OpenMP scheduling, the mean BFS time deteriorates rapidly once reaching a certain σ. In figure 5 this occurs for $\log_2 \sigma > 17$. The reason for this is because we are using 64 ($= 2^6$) threads on the outer for-loop of algorithm 5 on a graph that has 2^{23} vertices. Remember that σ is the sorting degree, i.e. the number of consecutive vertices to sort together. When the graph used in the BFS has $n = 2^{23}$ vertices, and we are using 64 threads with static scheduling, each thread will process exactly $\frac{2^{23}}{2^6} = 2^{17} = 131.072$ vertices in a single matrix-vector multiplication. Once $\log_2 \sigma > 17$, sorting the vertices will now potentially move vertices to a group of vertices processed by a different thread. Specifically, vertices with higher degrees will be sorted into groups together, while vertices with low degree will be sorted into groups of vertices with lower degree. However, threads still process the same number of threads. The result is that once $\log_2 \sigma > 17$, some threads receive larger workloads to process than other threads, as they are performing the matrix-vector multiplication for vertices with higher degrees (thus more edges to process).

In general, this occurs when $\log_2 \sigma > \log_2 n - \log_2 \mathit{num\_threads}$. In figure 5, with σ set to full sorting ($\sigma = 2^{23}$), the mean time is really the time it takes a single thread to process the 131.072 vertices with highest degree. To deal with this issue, we propose two solutions.

The first is to compute the optimal σ beforehand, in our case it would be $\log_2 \sigma_{opt} = \log_2 n - \log_2 \mathit{num\_threads}$. This can be seen in figures 5 and 10.

The second solution is to use dynamic OpenMP scheduling. We show this in figure 5. The effect is that during runtime, threads are given a single iteration of the for-loop to process before returning for another, instead of dividing the vertices into predetermined groups. Thus, the workloads are more evenly distributed. However, there is a slight overhead for supporting the dynamic scheduling which causes the dynamic scheduling to perform slightly worse than using static scheduling with the optimal choice of σ.

With regards to the parallel performance, we notice from figures 8a and 8b that BFS with CRS scales better than with SlimSell. One explanation for this could be the result of the selected σ. At the time of the tests, we were not considering an optimal σ for SlimSell. The results from figure 7 showing the effect of increasing number of threads, we had a constant $\sigma = 17$ while the number of threads varied. Notice how the parallel efficiency is very similar for CRS and SlimSell when the number of threads is 64. This could be because we have concluded that $\sigma = 17$ is optimal for the graph size 64 threads. It would be interesting to rerun these tests but where an optimal σ is selected for each number of threads. In general, one of the main factors that affect the parallel performance is how well the work is divided over threads. In our case, some chunks of SlimSell are much longer than others. For example, in our test graph with $n = 2^{23}$, $\bar{\rho} = 16$, the largest vertex degree is 257.623, while other vertices have 0 edges. In our implementation, this is a limiting factor for the speedup. One way to solve this could be to parallelize the inner for-loop of algorithm 5 so that no single thread must process very large chunks.

8 Conclusion and future work

Using vectorization is a viable method for improving the overall performance of breadth-first search. When using SlimSell, selecting optimal σ allows for better use of the available hardware that modern systems often support. The SlimSell format allows for efficient use of SIMD-instructions by avoiding the need for horizontal reduction or a way of dealing with loop remainders that attempting to vectorize the CRS format results in. Using SlimSell we show how it can outperform other formats, such as the common CRS sparse matrix format.

However, when using the SlimSell format with multi-threaded systems, it is important to remember that setting σ to the maximum might not be optimal if the workloads are statically divided over the threads. Testing the parallel performance with an optimal choice of σ should be done to determine the effects of σ with regards to parallel efficiency.

For future studies, it would interesting to see how the permutation of the vertices affects the performance of the BFS. Currently, SlimSell depends on two variables σ and C. However, the SIMD-instruction to load the right-hand side vector could potentially be improved by finding a permutation of the vertices such that memory accesses to a particular data are grouped. Could one for instance introduce another parameter that groups vertices based on the number of common neighbours for vertices?

References

[1] Andrew Lumsdaine et al. “Challenges in parallel graph processing”. In: *Parallel Processing Letters* 17.01 (2007), pp. 5–20.

[2] Maciej Besta et al. “Slimsell: A vectorizable graph representation for breadth-first search”. In: *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE. 2017, pp. 32–41.

[3] Daniel Kusswurm. “Advanced Vector Extensions (AVX)”. In: *Modern X86 Assembly Language Programming: 32-bit, 64-bit, SSE, and AVX*. Berkeley, CA: Apress, 2014, pp. 327–349. ISBN: 978-1-4842-0064-3. DOI: 10.1007/978-1-4842-0064-3_12. URL: https://doi.org/10.1007/978-1-4842-0064-3_12.

[4] Charles R Harris et al. “Array programming with NumPy”. In: *Nature* 585.7825 (2020), pp. 357–362.

[5] David A Bader and Kamesh Madduri. “Designing multithreaded algorithms for breadth-first search and st-connectivity on the Cray MTA-2”. In: *2006 International Conference on Parallel Processing (ICPP’06)*. IEEE. 2006, pp. 523–530.

[6] Jeremy Kepner and John Gilbert. *Graph algorithms in the language of linear algebra*. SIAM, 2011.

[7] Jeremy Kepner et al. “Graphs, Matrices, and the GraphBLAS: Seven Good Reasons”. In: *Procedia Computer Science* 51 (2015). International Conference On Computational Science, ICCS 2015, pp. 2453–2462. ISSN: 1877-0509. DOI: https://doi.org/10.1016/j.procs.2015.05.353. URL: http://www.sciencedirect.com/science/article/pii/S1877050915011618.

[8] *Graph 500*. URL: https://graph500.org/. (accessed: 2020-12-04).

[9] Moritz Kreutzer et al. “A unified sparse matrix data format for efficient general sparse matrix-vector multiplication on modern processors with wide SIMD units”. In: *SIAM Journal on Scientific Computing* 36.5 (2014), pp. C401–C423.

[10] Alexander Monakov, Anton Lokhmotov, and Arutyun Avetisyan. “Automatically Tuning Sparse Matrix-Vector Multiplication for GPU Architectures”. In: *High Performance Embedded Architectures and Compilers*. Ed. by Yale

N. Patt et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 111–125. ISBN: 978-3-642-11515-8.

[11] *Intel Intrinsics*. URL: `https://software.intel.com/sites/landingpage/IntrinsicsGuide/`. (accessed: 2021-01-4).

[12] *CRAY CX40 Beskow, PDC*. URL: `https://www.pdc.kth.se/hpc-services/computing-systems/beskow-1.737436`. (accessed: 2020-12-04).

A Code listings

```
    // create a vector of repeating (8 or 4) 32-bit integers from argument a
    _mm256i _mm256_set1_epi32(int a)
    __m128i _mm_set1_epi32(int a)

    // set (8 or 4) 32-bit integers with specified values
    _mm256i _mm256_set_epi32(int e7, int e6, ..., int e0)
    __mm128i _mm_set_epi32(int e3, int e2, int e1, int e0)

    // loads 256/128-bits of of integer data from memory
    __m256i _mm256_loadu_si256(__m256i const* mem_adr)
    __m128i _mm_loadu_si128(__m128i const* mem_adr)

    // store 256/128-bits (of 32-bit integers) in a to memory
    __m256i _mm256_storeu_si256(void* mem_adr, __m256i a)
    __m128i _mm_storeu_si128(void* mem_adr, __m128i a)

    // compare 32-bit integers in a and b for equality, return the result
    __m256i _mm256_cmpeq_epi32(__m256i a, __m256i b)
    __m128i _mm_cmpeq_epi32(__m128i a, __m128i b)

    // sets 8-bits from a or b depending on mask
    __m256i _mm256_blendv_epi8(__m256i a, __m256i b, __m256i mask)
    __m128i _mm_blendv_epi8(__m128i a, __m128i b, __m128i mask)

    // find the min 32-bit integers between a and b
    __m256i _mm256_min_epi32(__m256i a, __m256i b)
    __m128i _mm_min_epi32(__m128i, __m128i b)
```

Listing 2: Intel Instrinsics

```cpp
void tropical_SlimSell_MVM_w8(
    std::vector<int32_t>& y,
    const sellcs& g, // graph in SlimSell format
    const std::vector<int32_t>& x) {

    // vectors with 1, -1 and nverts, respecitively
    __m256i ones = _mm256_set1_epi32(1);
    __m256i m_ones = _mm256_set1_epi32(-1);
    __m256i infs = _mm256_set1_epi32(g.nverts);

    // iterate over each chunk
    // we can use an omp pragma here to divide the work over threads
    for(int32_t i = 0; i < g.n_chunks; ++i) {
        __m256i tmps, col, vals, rhs;
        int32_t c_offs;
        c_offs = g.cs[i];

        // load chunk from rhs vector x
        tmps = _mm256_loadu_si256((__m256i*)&x[i*g.C]);

        // iterate over each column of the current chunk
        for(int32_t j = 0; j < g.cl[i]; ++j) {
            // load column j of current chunk
            col = _mm256_loadu_si256((__m256i*)&g.cols[c_offs]);

            // identify which values are padding
            vals = _mm256_cmpeq_epi32(m_ones,col);

            // set padded values to inf, otherwise 1
            vals = _mm256_blendv_epi8(ones,infs,vals);

            // load the corresponding values in the rhs vector
            rhs = _mm256_set_epi32( x[g.cols[c_offs+7]],
                                    x[g.cols[c_offs+6]],
                                    x[g.cols[c_offs+5]],
                                    x[g.cols[c_offs+4]],
                                    x[g.cols[c_offs+3]],
                                    x[g.cols[c_offs+2]],
                                    x[g.cols[c_offs+1]],
                                    x[g.cols[c_offs+0]] );

            // compare the current path length (1+rhs) with the current minimum
            tmps = _mm256_min_epi32(_mm256_add_epi32(rhs,vals),tmps);
            c_offs += g.C;
        }
        // store the result of the processed chunk
        _mm256_storeu_si256((__m256i*)&y[i*g.C], tmps);
    }
}
```

Listing 3: BFS-SlimSell with Intel Intrinsics and width 8.