# High Performance Parallel Summed-Area Table Kernels for Multi-Core and Many-Core Systems

Angelos Papatriantafyllou[1] and Dimitris Sacharidis[2]

[1] Faculty of Informatics, Institute of Information Systems, Research Group Parallel Computing, TU Wien, Vienna, Austria
papatriantafyllou@par.tuwien.ac.at

[2] Faculty of Informatics, Institute of Software Technology and Interactive Systems, E-Commerce Group, TU Wien, Vienna, Austria
dimitris@ec.tuwien.ac.at

**Abstract.** The summed-area table (SAT), also known as integral image, is a data structure extensively used in computer graphics and vision for fast image filtering. The parallelization of its construction has been thoroughly investigated and many algorithms have been proposed for GPUs. Generally speaking, state-of-the-art methods cannot efficiently solve this problem in multi-core and many-core (Xeon Phi) systems due to cache misses, strided and/or remote memory accesses. This work proposes three novel cache-aware parallel SAT algorithms, which generalize parallel block-based prefix-sums algorithms. In addition, we discuss 2D matrix partitioning policies which play an important role in the efficient operation of the cache subsystem. The combination of a SAT algorithm and a partition is manually tuned according to the matrix layout and the number of threads. Experimental evaluation of our algorithms on two NUMA systems and Intel's Xeon Phi, and for three datatypes (int, float, double) by utilizing all system cores, shows, in all experimental settings, better performance compared to the best known CPU and GPU approaches (up to 4.55× on NUMA and 2.8× on Xeon Phi).

## 1 Introduction

The construction of a summed-area table (SAT) is a well-studied problem in computer graphics and vision [1, 13, 14], with applications in texture filtering. Since first introduced by Crow [4], several parallel implementations for GPUs [6, 9, 7] have been proposed. Given a matrix $x$ of size $n \times m$, where $n$, $m$ are the number of rows and columns, respectively, the problem is to compute the sum of all elements $x(i, j),\ 0 \leq i < n$ and $0 \leq j < m$, according to the formula:

$$y(i,j) = \sum_{0 \leq r < i} \sum_{0 \leq c < j} x(r,c) \tag{1}$$

SAT is the 2D generalization of the prefix-sums, or scan, problem, whose parallelization has also been extensively studied; the prefix-sums problem (assuming + as the associative operation) for an array $x$ of length $n$ is to compute the sums

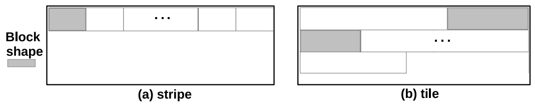

**Fig. 1.** The 2D matrix partitioning policies: `stripe` (left), and `tile` (right). The grey box indicates the shape of a block according to each partitioning

$y(i) = \sum_{0 \le r < i} x(r)$. Hence, a straightforward method to parallelize SAT is to scan the matrix row by row, and for each row apply a parallel scan kernel from the literature [5, 10–12] to compute its prefix-sums in-place, and then perform in parallel vectorized additions with the prefix-sums of the previous row. Such an approach has throughput bounded by the column size; each process unit (core, warp), henceforth termed thread, operates on a block of size $\frac{m}{p}$, where $p$ is the number of threads. Hence, better performance is achieved with 2D blocks.

In literature, there have been several 2D block-based parallel SAT algorithms for GPUs (Hensley *et al.* [6], Kasagi *et al.* [7], Nehab *et al.* [9] and Yan *et al.* [15]) and CPUs (Zhang [16]). These can be classified to those that perform within a block prefix-sums along a single dimension [6, 15, 16], and those along both dimensions, [9, 7]. Methods of the former use blocks of size $\frac{n}{p} \times m$ (or $\frac{m}{p} \times n$), resulting in performance degradation when run on NUMA multi-core systems, due to cache misses since blocks can be bigger than the LLC (Last Level Cache), and remote memory accesses imposed by the block assignment, explained later in the related work. On the other hand, algorithms of the latter use square $b \times b$ blocks, and could thus suffer from strided memory accesses when row-major matrix allocation is used on multi-core systems. In addition, the degree of parallelism of [7] is bounded by the number of blocks of each anti-diagonal, which ranges from 1 to $\frac{min(n,m)}{b} - 1$.

**Contributions**. Our research is motivated by two important facts: 1) the performance limitations of square blocks used in [7, 9], and 2) the lack of performance and scalability in current parallel SAT implementations for multi-core systems. Both [7, 9] execute row- and column-wise prefix-sums, where the latter are expensive, w.r.t. performance, due to strided memory accesses. To alleviate this overhead, we propose to use non-square blocks of size $br \times bc$, where $br$, $bc$ are the number of rows and columns, respectively, that are horizontally "stretched" (while being vertically "squeezed"), i.e., $bc > br$. The in-parallel processed blocks must fit into the LLC; thus, $p \times br \times bc$ should not exceed the LLC size.

For the following discussion it helps to conceptually represent the matrix as a sequence of stripes each having size $br \times m$. Depending on the value of $bc$ relative to $m$, we distinguish two partitionings. In the first called `stripe`, $bc$ is equal to $\frac{m}{p}$, which means that all blocks processed in parallel lie within a stripe. In the second and more general called `tile`, $bc \neq \frac{m}{p}$, meaning that a block can span over two consecutive stripes, and thus correspond to a non-rectangular area of the matrix. Fig. 1 depicts the two partitionings on the actual matrix, and illustrates that a block in `tile` can be non-rectangular (shaded).

In addition, we propose three cache-aware parallel SAT algorithms, called `psat_cpps`, `psat_mcstl` and `psat_sarpps`, which process square and non-square blocks, and are generalizations of three parallel block-based prefix-sums algorithms: CPPS [10], MCSTL [12], and the method of Chatterjee *et al.* [3], which we henceforth call SARPPS (Scan after Reduction Parallel Prefix-Sums). In particular, our algorithms operate in three phases: Phases 1 and 3 are devoted to in-block computations, and in Phase 2 shared data are propagated across the threads. They differ in the tasks performed in Phases 1 and 3, which are optimized (e.g., using vectorization, loop-unrolling) based on the target system and datatype. Depending on the matrix partitioning employed, certain tasks of Phase 2 are omitted resulting in distinct performance behavior for the same algorithm.

We carefully evaluate the performance and scalability of our kernels compared to the state-of-the-art method for GPU by Kasagi *et al.* [7], and CPU by Zhang [16]. We experiment on two NUMA systems (Westmere- and Opteron-based) and Intel's Xeon Phi, considering different datatypes (int, float, double), while varying number of threads. Our results verify the limitations of existing work when deployed in multi-core and many-core systems. In particular, when utilizing almost the maximum available physical cores, the speedup of [7] drops, and in certain settings down-scale behavior is observed. On the other hand, [16] has the worst performance by a large margin in NUMA systems. In contrast, our proposed kernels have consistent performance behavior across all systems and datatypes, without requiring any modifications to the parallelization code, and in almost all cases outperform the competitors. Specifically, our kernels have up to 4.55x and 3.25x more speedup in NUMA systems, and up to 1.5x and 2.8x in the Xeon Phi system, compared to [16] and [7], respectively. Furthermore, we study the main parameters (e.g., block row and column size, row- and column-wise optimizations, matrix partitioning) and draw conclusions on how they influence the performance of our kernels. In summary, the performance gains we measure are due to the better utilization of the LLC, which is the direct result of larger (non-rectangular) blocks that could not be exploited by previous works.

The rest of this paper is organized as follows. Section 2 describes and reviews related works. Section 3 presents our algorithms in detail. Section 4 shows performance and scalability results. Section 5 concludes the paper.

## 2 Related Work

**Parallel prefix-sums kernels.** CPPS [10], MCSTL [12], and SARPPS [3] are block-based algorithms that solve the prefix-sums problem in parallel by splitting the input array in $p$ or $p+1$ blocks. Each block is processed in three phases, where the first and third perform in-block computations, while the second propagates shared data across threads. In Section 3, we discuss their generalization to SAT.

**Parallel SAT kernels.** There are two algorithmic classes for parallelizing SAT. While, they both split the matrix in blocks assigned to individual threads, they differ in their in-block computations. The first class, termed 1D, is to perform either row- or column-wise prefix-sums, while the second, termed 2D, is to perform prefix-sums in both dimensions, i.e., compute the block's SAT.

Previous 1D approaches on GPUs (Hensley *et al.* [6] and Yan *et al.* [15]) use 2-phase algorithms to compute in parallel first all horizontal and later all vertical prefix-sums. In [6], they use recursive doubling for computing the prefix-sums of both phases, whereas, in [15] they exploit vectorization over a column-major order input matrix by using an auxiliary $n \times m$ matrix. Similarly, Zhang [16] present a 1D method for multi-core systems. Noticeably, the performance of all 1D approaches is expected to be penalized on multi-core systems due to cache-unawareness, because blocks may exceed the LLC. Particularly on NUMA systems, it is expected more performance degradation due to NUMA-unawareness, since the data needed in Phase 2 may reside in remote NUMA nodes.

Nehab *et al.* [9] present a generalization of the SARPPS algorithm for GPUs, which works quite similar to our `psat_sarpps` algorithm. The input matrix is partitioned in blocks of size $b \times b$, where $b$ matches the *warp* size of an SM (i.e. $b = 32$). Each *warp* computes the SAT of a block with the help of the last block row and block column of the other *warp*'s. Their approach could compose and process also rectangular blocks but it is unable to exploit non-rectangular blocks.

Kasagi *et al.* [7] present a different 2D approach, called 1R1W, where each matrix element is read from and written to DRAM only once. To accomplish that, an $n \times n$ matrix is partitioned into $\frac{n}{b} \times \frac{n}{b}$ blocks of equal $b \times b$ size, where $b \leq \frac{n}{p}$, and processed anti-diagonally in $\frac{2n}{b} - 1$ steps. In the $k$th step, where $0 \leq k < \frac{2n}{b} - 1$, there are $\min(p, k+1)$ active threads, where each computes its own block by using elements of the adjacent left, top and diagonal blocks. Before entering each step, the threads must synchronize at a global barrier. This approach is expected to exhibit poor performance and scalability in NUMA systems for very large $p$ due to two reasons. First, while increasing $p$ more NUMA nodes are activated and subsequently, more LLCs can be utilized. However, in this approach the block size is bounded by $n$, implying that the increase of $p$ leads to smaller block sizes and eventually, the LLC footprint is reduced. Exploiting less LLC footprint means that the CPUs process faster the cached blocks and the kernel's performance is exposed to the main memory latency overhead. Second, while increasing $p$ it would need more synchronization steps to reach the maximum degree of parallelism.

**In-block optimizations.** Both 1D and 2D approaches benefit from prefix-sums optimizations. Zhang [16] optimizes the row-wise prefix-sums with an algorithm called *enhanced parallelism*, which breaks each row in groups of 6 elements; each group is computed by independent prefix-sums pairs. However, such a technique is not efficient for large vector widths machines like Xeon Phi due to its lack of leveraging vector instructions. Previous works on GPUs [6, 9, 15] exploit the SIMT (Single Instruction Multiple Thread) model in order to vectorize the row-wise prefix-sums. Unfortunately, SIMT is not compatible with the SIMD (Single Instruction Multiple Data) programming model used in CPUs.

A SIMD-based approach is provided by OpenCV (Open Source Computer Vision) [2], which is a computer vision and machine learning library. This ap-

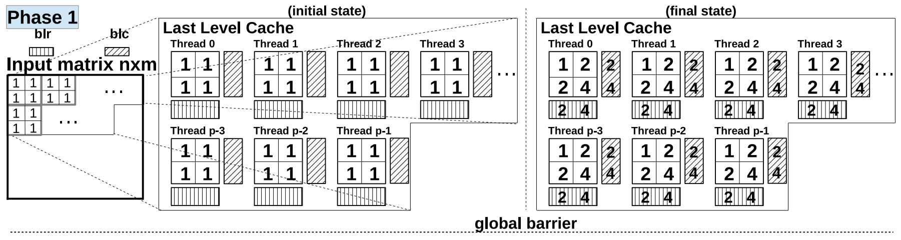

**Fig. 2.** Phase 1: each of the $p$ threads is assigned a distinct $b \times b$ block of the $n \times m$ matrix and runs a sequential SAT kernel. The computed last block row and block column are copied into the `blr` and `blc` shared buffers, respectively. All the blocks fit into the system's LLC. At the end of this phase, thread 0 has completed its block

proach uses the SSE instruction set, which has a limited target group and works under specific input constraints (short numbers). Thus, it is not suitable for Xeon Phi systems and it is not applicable to arbitrary datatypes.

From the above discussion, it becomes clear that new optimizations techniques are necessary for multi-core systems. We remark that the sequential (in-block) SAT computation in all 2D methods is based on only one of the three known basic approaches for computing SAT [8]. In Section 3, we optimize this approach by further segmenting the assigned block to leverage deep cache hierarchies, investigate prefix-sums optimizations presented in [10], and also explore the optimization space of another basic SAT approach.

## 3 Parallel Summed-Area Table Kernels (PSAT)

Our algorithms construct the SAT of a matrix $x$ in-place, with an associative operator $+$ over the basic type, by splitting $x$ in 2D blocks. The blocks are formed according to `stripe` and `tile` partitionings, and are further tuned to meet load-balancing requirements according to each PSAT kernel, later explained. Each block is assigned only once to a unique thread and processed in-cache across some or all the algorithmic phases, as illustrated in Fig. 2–4. In addition, the threads use two shared buffers, called `blr` and `blc`, to propagate their last row and column, respectively, across all threads. For simplicity and readability, we use square blocks to describe our implementations.

### 3.1 Implementations

**Psat_cpps**. This algorithm is the generalization of CPPS [10], which works as follows: 1) Each thread computes the prefix-sums for its assigned block, 2) an inclusive scan is computed over the last element of all the assigned chunks, and 3) each thread, except for thread 0, propagates a previous corresponding cumulative sum to its own elements; thread 0 is assigned and process a new block.

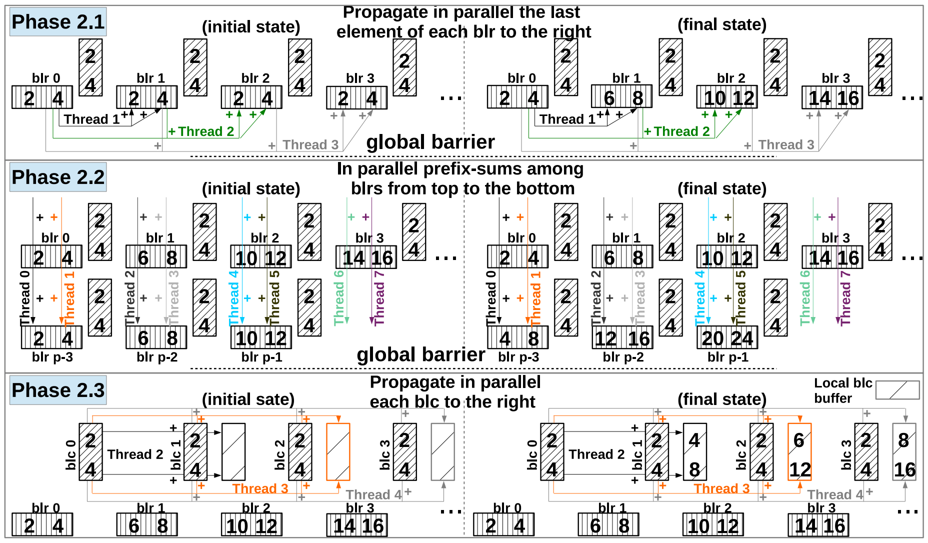
Phase 2.1
(initial state)
Propagate in parallel the last element of each blr to the right
(final state)
global barrier
Phase 2.2
In parallel prefix-sums among blrs from top to the bottom
global barrier
Phase 2.3
(initial sate)
Propagate in parallel each blc to the right
(final state)
Local blc buffer

**Fig. 3.** Phase 2: shared data propagation. Each thread collects the cumulative sum of the last element of each `blr` at its left and propagates it to its own `blr` (Phase 2.1). Subsequently, in parallel column-wise prefix-sums are computed by adding the elements of all the `blr`'s, from top to bottom, including the last `blr`'s from the previous group of cached blocks (Phase 2.2). In Phase 2.3, each thread computes and stores locally the cumulative sums of the `blc`'s at its left

In Phase 1 of `psat_cpps` (Fig. 2), each thread is assigned a $b \times b$ block, on which a sequential SAT is invoked. At the completion of Phase 1, thread 0 is the only thread that has successfully computed its own block. Subsequently, each thread copies the last computed block row and block column to the shared buffers `blr` and `blc`, respectively, and synchronizes with the others at a global barrier. The role of these buffers is crucial. They hold the fragments of the final product computed later at Phase 3. For instance, thread $p-1$ needs to sum up all the information from the `blc`'s and `blr`'s belonged to the threads accessing the matrix rows and matrix columns which are equal and less than $p-1$'s.

Phase 2 (Fig. 3) is devoted to process and propagate the shared data from the buffers `blr` and `blc` across threads. The phase is split in three parts, where the first two are responsible to process the `blr`'s, and the third part handles to process the `blc`'s. Therefore, each thread computes the cumulative sum of the last element of the `blr`'s related to the threads at its left. Subsequently, the sum is used to propagate the elements of its own `blr` (Phase 2.1). Then, the threads synchronize once again before passing to Phase 2.2, where the aggregation of each shared-row `blr`'s is split in $p$ segments of size $\frac{m}{p}$ and each segment is assigned to each thread. In total, each thread has to process $\frac{pb}{m}$ segments, on which is computed the column-wise inclusive prefix-sums by adding the $i$th element, $i = 0, 1, ..., \frac{m}{p} - 1$, of the $j$th segment, $j = 0, 1, ..., \frac{pb}{m} - 1$, with the $i$th element of

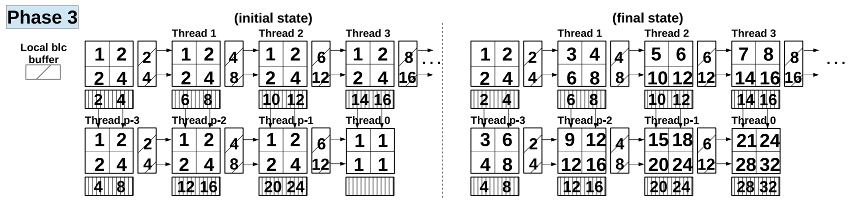

**Fig. 4.** Phase 3: The adjacent top and left buffers are used by each thread to update its block. Thread 0 is assigned a new block and computes the SAT product

the $j+1$th segment, and so on. The threads synchronize again before accessing Phase 2.3, where each thread computes the reduction of the elements of the `blc`'s of the threads located at its left by adding the $i$th element, $i = 0, 1, ..., b-1$ of the $j$th `blc` with the $i$th element of the $j+1$th `blc`, $j$ starts with the id of the thread being the first at its left and ends with the id of the previous thread. It is implied that a barrier is unnecessary at this point. The reductions are stored in local buffers, and are accessed in Phase 3 locally by their associate thread.

Finally, in Phase 3 (Fig. 4), each thread uses its local `blc` and the updated `blr` (top) to construct the table by propagating their values to its block elements. In addition, thread 0 is assigned a new block, and invokes a sequential SAT kernel by adding at the same time the elements of the left and top buffers. After careful benchmarking, this new block must be a factor of 4 smaller; the block is fetched from DRAM while the other threads work on cached data.

**Psat_sarpps**. SARPPS [3] generalizes `psat_sarpps` kernel, and works as follows: 1) each thread first acquires the total sum of its elements, 2) an inclusive scan is performed over the previous sums, and 3) each thread computes the prefix-sums of its elements by adding first the corresponding sum from Phase 2.

In Phase 1, the threads do not invoke a sequential SAT but a 2D reduction kernel, which computes the sums of every block row and block column by storing the sum of the $i$th block row, $i = 0, 1, ..., b-1$, into the $i$th cell of the `blc` buffer and the $j$th block column, $j = 0, 1, ..., b-1$, into the $j$th cell of the `blr` buffer. Subsequently, in Phase 2, the threads are grouped according to which of them access the same matrix rows. The threads assigned non-rectangular blocks (refer to `tile` in Fig. 1) are potential members of two groups. However, we do not permit "double membership", and we place those threads to the groups with the least members. Consecutively, each group runs a parallel prefix-sums on its elements by invoking CPPS [10], configured with a two-level-nested-loop sequential prefix-sums kernel, described in [10]. Phases 2.2 and 2.3 are executed as before (refer to `psat_cpps`). In Phase 3, each thread updates its first block column and row by using the computed data of its local `blc` and the shared `blr` (left), respectively, and subsequently it executes a sequential SAT kernel.

**Psat_mcstl**. This algorithm is the generalization of the MCSTL [12] algorithm, which resembles SARPPS. However, they are distinguishable from each other. In MCSTL's Phase 1, thread 0 computes the prefix-sums of a block, which differs from the block that it computes its prefix-sums in Phase 3.

In Phase 1 of **psat_mcstl**, a block of size $\frac{b}{2} \times \frac{b}{2}$ is assigned to thread 0 and $p-1$ blocks of size $b \times b$ to the other threads. Thread 0 invokes a sequential SAT kernel and the others a 2D reduction kernel for their blocks. The block of thread 0 is a factor of 4 smaller than the other blocks because of the heavyweight computation of the sequential SAT compared to the 2D reduction task. The results of the last computed block row and column are stored again in the `blr` and `blc` buffers, respectively. At that time, thread 0 has produced the final result for its block. Phase 2 can be executed as described in `psat_sarpps`. However, since thread 0 `blc` holds already the final values it should not take part in Phase 2. Instead, we re-segment the input of its group and assign a task to thread 0. Phases 2.2 and 2.3 are executed as before (refer to `psat_cpps`). Consecutively, Phase 3 operates as described in `psat_sarrps`, however, thread 0 employs a different block; for better load balancing this block size must be $b \times \frac{b}{2}$.

**Optimizations**. Due to space constraints, we briefly describe the optimizations that took place for improving the performance of the sequential SAT kernel used by our PSAT kernels. We have optimized two basic SAT approaches, described in [8]; one is used by [7, 9]. Since both approaches compute prefix-sums row-wisely, we first tested several optimized sequential prefix-sums kernels, presented in [10]. In addition, both approaches also compute the prefix-sums column-wisely, which can be easily vectorized. However, the size of the vectorized rows can be bigger than the underlying L2/L1 cache size. Thus, we considered performance improvements through further input segmentation. The new formed sub-blocks have 2–4 rows each, and the column size varies according to the architecture. We investigated improving the ILP and the spatial locality of the column-wise prefix-sums. The former through hard-coding the execution of 2–4 independent row-wise prefix-sums of each sub-block, and the latter by storing consecutively in memory the row-wise results (in vector-width chunks).

## 3.2 Matrix Partitionings

Conceptually, the matrix is composed by a sequence of stripes of size $br \times m$. Accordingly, our PSAT kernels split the stripes in groups of blocks whose cumulative size does not exceed the aggregated system's LLC. Subsequently, each group is processed in parallel by $p$ threads. In `stripe` partitioning (Fig. 1 (left)), the kernels process one stripe at a time by computing groups of blocks located in one stripe, so that, $bc \times p \leq m$. On the contrary, in `tile` (Fig. 1 (right)), the formed groups of blocks may span across several stripes, as long as $bc \leq m$.

`Stripe` affects the functionality of our algorithms. In particular, Phases 2.1 and 2.2 are omitted since all threads operate within the same stripe, where the column-wise propagation is omitted. The reason is that each thread is assigned the same matrix columns across different stripes, and thus can directly access the last block row of the previous stripe without needing the `blr` buffer.

Regarding the scalability of our kernels, we make the following observations. Increasing the number of threads ($p$) in `stripe` means that the block column size ($bc$) decreases resulting in the following trade-off: we can increase the $br$ (wider stripe) to improve the LLC utilization at the expense of multiple strided

**Table 1.** Specifications of the three systems (Mars, Saturn and MIC)

| System | CPU (model & freq.) | # cores | # cores/NUMA | # NUMA nodes | LLC/NUMA | L2 | L1 |
|---|---|---|---|---|---|---|---|
| Mars | Intel Xeon E7-8850 2.0 GHz | 80-hyperthreaded | 10 | 8 | 24576K | 256K | 32K |
| Saturn | AMD Opteron 6168 1.9 GHz | 48 | 6 | 8 | 5118K | 512K | 32K |
| MIC | Intel Xeon Phi 5110P 1.059 GHz | 60-hyperthreaded | - | - | only L2 cache | 512K | 64K |

memory accesses, or keep the LLC utilization low, thus avoiding this overhead (strided accesses). In NUMA systems, this effect is amplified since the LLC size increases by activating more NUMA nodes. The benefit of `stripe` over `tile` is the decrease of the amount of propagated shared data. Thus, we expect `stripe` to perform better for small $p$. On the contrary, `tile` is not affected by this trade-off and promises better scalability in both multi-core and many-core systems.

## 4 Evaluation

We implemented our PSAT kernels in C with Pthreads and tested on three systems: two NUMA systems, called Mars and Saturn, and one Intel's Xeon Phi, called MIC. Detailed system information are listed in Table 1. All benchmarks are compiled with Intel's ICC 14.0.1 with -O3 optimization level and executed 30 times. The results show median values. Due to space limitations, we report only results for a specific problem size; similar behavior is observed in other problem sizes. Each kernel is configured and tested with different sequential SAT kernels and block sizes. We only report results for the best configurations. The experiments show the performance and scalability behavior of our PSAT kernels, comparing against 1R1W [7] and Zhang [16]. For a fair comparison, we re-implemented 1R1W for x86 systems according to [7], tested with rectangular blocks, and applied x86-based sequential optimizations. For the testbeds, in-parallel first-touch page placement was applied and consecutive thread pinning.

**PSATs performance**. Fig. 5 (left) depicts the performance (execution time in seconds) of our PSAT kernels (`psat_cpps`, `psat_sarpps`, `psat_mcstl`). Each kernel has been separately tested with `tile` and `stripe` partitionings. In addition, Fig. 5 (right) depicts a breakdown phase analysis of our PSATs in `stripe` mode by reporting the sum of the execution time (in milliseconds) of Phases 1 and 3 when computing $p$ (= #threads) blocks. All the testbeds construct the SAT of a 12K×12K integer matrix on Mars with different number of threads. Due to quantitatively similar results, other systems and datatypes are omitted.

Fig. 5 (left) justifies our assumptions about the behavior of both partitionings independently of PSAT kernel. The `stripe` behaves better for small number of threads due to less computations, and the `tile` is better for large number of threads due to better cache utilization and less synchronization steps caused by handling bigger block sizes. In most cases, `psat_cpps` outperforms the other kernels, even though Phase 1 of `psat_sarpps` is faster, due to the reduction

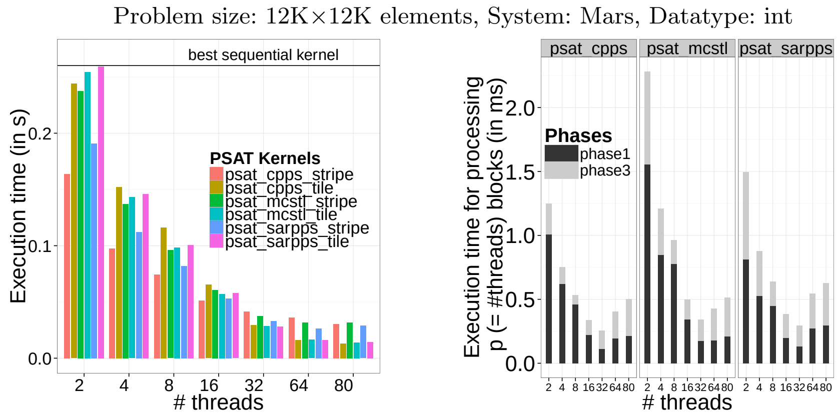

**Fig. 5.** Performance comparison (left) between our three PSAT kernels (`psat_cpps`, `psat_sarpps`, `psat_mcstl`) configured with the two partitionings (`tile`, `stripe`), and breakdown phase analysis (right) of our PSATs in `stripe` mode, which reports the sum of the execution time of Phases 1 and 3 for computing $p$ (= #threads) blocks. The testbeds run on Mars for different number of threads and a 12K×12K integer matrix

computations being auto-vectorized by gcc and icc compilers, as depicted in the breakdown analysis of Fig. 5 (right). Therefore, our analysis suggests that running a sequential SAT kernel first leads to better performance.

**PSATs speedup**. Fig. 6 depicts the absolute speedup comparison between 1R1W, Zhang and our best performing PSAT for each of the `tile` and `stripe` partitionings. The results are collected by all our systems after composing the SAT of a 12K×12K matrix separately for integers, floats and doubles. In addition, the best performing sequential kernel that we have is selected as a baseline.

Fig. 6 shows that in all cases our kernels outperform 1R1W and Zhang for large number of threads ($p$) in `tile`. In particular, our kernels run 1.2×–3.25× faster than 1R1W (all datatypes) with all system cores: 1) 3.25× in Mars, 2) 1.8× in Saturn and 3) 2.8× in MIC. Nonetheless, 1R1W behaves slightly better for small $p$ since it spends zero time on processing shared data, and needs fewer synchronization steps until all threads are utilized in parallel. Regarding the performance of Zhang in MIC, it is almost equal to that of our best kernel for integers, but it is 1.5× and 1.2× slower for floats and doubles, respectively. In addition, Zhang proves to be inefficient in NUMA systems, for reasons discussed in Section 2. For instance, with all system cores, Zhang is slower by 3.25× (integers and floats) and 3.5× (doubles) on Mars, and 4.55× (integers and floats), and 4.15× (doubles) on Saturn. In conclusion, we observe that the best performance of our kernels, in `tile` with all cores, is achieved when the block column size ($bc$) is smaller than the matrix column size by 1.67× (integers and floats) and 3.3× (doubles) in Mars, 4× (all datatypes) in Saturn and 5× (doubles and floats) and 10× (integers) in MIC. In addition, the block row size is by far smaller than the $bc$ in NUMA (57×–114×), and still quite smaller in MIC (1.2×–4.8×).

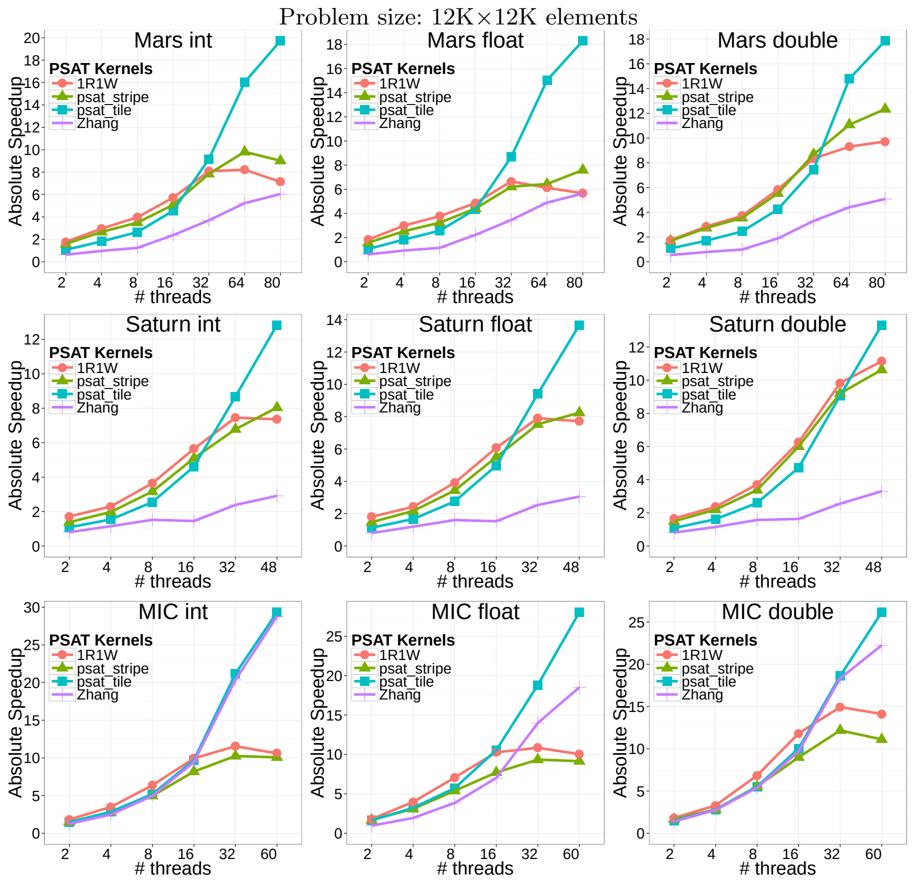

**Fig. 6.** Absolute speedup comparison between 1R1W kernel [7], Zhang's [16], and our best performing PSAT for each of the two partitionings: `tile` and `stripe` running on all systems. The kernels construct a 12K×12K matrix for three datatypes (int, float, double). The best performing sequential kernel that we have is selected as a baseline

## 5 Conclusions and Future Work

In this paper, we present three new cache-aware parallel SAT (Summed-Area Table) algorithms, called `psat_cpps`, `psat_sarpps` and `psat_mcstl`, for many-core and multi-core systems. Our algorithms are generalizations of three block-based parallel prefix-sums algorithms, and can process rectangular and non-rectangular blocks in order to utilize better the cache subsystem. We provide performance and speedup results after comparing against Kasagi *et al.* [7] and Zhang [16]. Our next step will be designing an auto-tuning mechanism capable of finding the best configurations (parallel and sequential SAT, matrix partitioning, etc.) for different problem sizes in order to increase system efficiency.

## 6 Acknowledgements

We would like to thank the members of the TU Wien Research Group Parallel Computing and the anonymous reviewers for their valuable comments.

## References

1. Bay, H., Ess, A., Tuytelaars, T., Gool, L.J.V.: Speeded-Up Robust Features (SURF). Computer Vision and Image Understanding 110(3), 346–359 (2008)
2. Bradski, G.R., Kaehler, A.: Learning OpenCV - Computer Vision with the OpenCV Library: Software That Sees. O'Reilly (2008)
3. Chatterjee, S., Blelloch, G.E., Zagha, M.: Scan Primitives for Vector Computers. In: Proceedings Supercomputing'90. pp. 666–675 (1990)
4. Crow, F.C.: Summed-Area Tables for Texture Mapping. In: Proceedings of the 11th Annual conference on Computer Graphics and Interactive Techniques (SIGGRAPH). pp. 207–212 (1984)
5. Dotsenko, Y., Govindaraju, N.K., Sloan, P., Boyd, C., Manferdelli, J.: Fast Scan Algorithms on Graphics Processors. In: Proceedings of the 22nd Annual International Conference on Supercomputing (ICS). pp. 205–213 (2008)
6. Hensley, J., Scheuermann, T., Coombe, G., Singh, M., Lastra, A.: Fast Summed-Area Table Generation and its Applications. Computer Graphics Forum 24(3), 547–555 (2005)
7. Kasagi, A., Nakano, K., Ito, Y.: Parallel Algorithms for the Summed Area Table on the Asynchronous Hierarchical Memory Machine, with GPU Implementations. In: Proceedings of the 43rd International Conference on Parallel Processing (ICPP). pp. 251–260 (2014)
8. Kisacanin, B.: Integral Image Optimizations for Embedded Vision Applications. In: Proceedings of the 2008 IEEE Southwest Symposium on Image Analysis and Interpretation (SSIAI). pp. 181–184 (2008)
9. Nehab, D., Maximo, A., Lima, R.S., Hoppe, H.: GPU-Efficient Recursive Filtering and Summed-Area Tables. ACM Transactions on Graphics 30(6), 176 (2011)
10. Papatriantafyllou, A.: Energy Characterization and Optimization of Parallel Prefix-Sums Kernels. In: Euro-Par 2015: Parallel Processing Workshops - Euro-Par 2015 International Workshops (2015)
11. Sengupta, S., Harris, M., Garland, M.: Efficient Parallel Scan Algorithms for GPUs. Tech. rep., NVIDIA Corporation (2008)
12. Singler, J., Sanders, P., Putze, F.: MCSTL: The Multi-Core Standard Template Library. In: Proceedings of the 13th International Euro-Par Conference on Parallel Processing. pp. 682–694 (2007)
13. Viola, P.A., Jones, M.J.: Rapid Object Detection using a Boosted Cascade of Simple Features. In: Proceedings of the 2001 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR). pp. 511–518 (2001)
14. Viola, P.A., Jones, M.J., Snow, D.: Detecting Pedestrians Using Patterns of Motion and Appearance. International Journal of Computer Vision 63(2), 153–161 (2005)
15. Yan, S., Zhang, Y., Long, G.: Summed-area Table Algorithm Optimization Based on the OpenCL. In: Proceedings of the ATIP/A*CRC Workshop on Accelerator Technologies for High-Performance Computing: Does Asia Lead the Way? (2012)
16. Zhang, N.: Working towards Efficient Parallel Computing of Integral Images on Multi-Core Processors. In: Proceedings of the 2nd International Conference on Computer Engineering and Technology (ICCET). pp. V2–30–V2–34 (2010)