3.5-D Blocking Optimization for Stencil Computations on Modern CPUs and GPUs

Anthony Nguyen, Nadathur Satish, Jatin Chhugani, Changkyu Kim and Pradeep Dubey
Throughput Computing Lab
Intel Corporation
Email: {anthony.d.nguyen,nadathur.rajagopalan.satish,jatin.chhugani,changkyu.kim,pradeep.dubey}@intel.com

Abstract—Stencil computation sweeps over a spatial grid over multiple time steps to perform nearest-neighbor computations. The bandwidth-to-compute requirement for a large class of stencil kernels is very high, and their performance is bound by the available memory bandwidth. Since memory bandwidth grows slower than compute, the performance of stencil kernels will not scale with increasing compute density. We present a novel 3.5D-blocking algorithm that performs 2.5D-spatial and temporal blocking of the input grid into on-chip memory for both CPUs and GPUs. The resultant algorithm is amenable to both thread-level and data-level parallelism, and scales near-linearly with the SIMD width and multiple-cores. Our performance numbers are faster or comparable to state-of-the-art-stencil implementations on CPUs and GPUs. Our implementation of 7-point-stencil is 1.5X-faster on CPUs, and 1.8X faster on GPUs for single-precision floating point inputs than previously reported numbers. For Lattice Boltzmann methods, the corresponding speedup number on CPUs is 2.1X.

I. Introduction

Stencil computation (SC) is used for a wide range of scientific and engineering disciplines [1], [2], [3], [4], [5], [6]. For a large class of SC kernels, the bandwidth-to-compute requirement is very high as a large number of bytes is needed for the few calculations. Furthermore, SC sweeps over the entire grids multiple times, evicting data from caches for 3D or higher-dimensional grids where the working set is too large even for the last-level cache.

On the other hand, compute capacity has increased through core counts and wider vector (SIMD) units. Core counts will increase rapidly as the number of on-chip transistors continue to grow. The SIMD width of modern CPU and GPU processors has been steadily increasing from 128-bit in SSE architectures, 256-bit in AVX [7] to 512-bit in the upcoming Larrabee [8] architecture. GPUs have a logical 1024-bit SIMD with physical SIMD widths of 256-bits on the Nvidia GTX 200 series, increasing to 512-bits on the upcoming Fermi [9] architecture. However, memory bandwidth is increasing at a slower pace than compute. Algorithms that are bound by memory bandwidth will not scale well to future architectures. Therefore, the high memory bandwidth requirement for SC makes it especially challenging to utilize the compute density of current and upcoming architectures.

In this paper, we present a novel 3.5D blocking algorithm that performs a 2.5D spatial blocking and an additional temporal blocking of the input grid into on-chip memory. Our 2.5D spatial blocking blocks in two dimensions and streams through the third dimension, thereby increasing the blocking size and resulting in better utilization of the on-chip memory. The temporal blocking performs multiple time steps of SC, re-using the data blocked into on-chip memory (caches for CPUs and shared memory and register files for GPUs). This reduces the effective bandwidth requirement and allows for full utilization of the available compute resources. The resultant algorithm is amenable to both thread-level and data-level parallelism, and scales near-linearly with SIMD width and multiple cores.

We apply this 3.5D blocking algorithm to two specific examples. The first example is a 7-point stencil for 3D grids. It comprises of 7 points: 1 point for the local grid site and 2 adjacent points for each of the x, y, and z dimensions. The second example uses a Lattice Boltzmann method (LBM). LBM is a class of computational fluid dynamics capable of modeling complex flow problems. It simulates the evolution of particle distribution functions over a 3D lattice over many time steps. Since LBM traverses the entire lattice in each time step to perform particle collisions and propagations, it accesses a large amount of data. The performance of previously optimized implementations of both 7-point stencil and LBM are bandwidth bound on state-of-the-art CPUs and GPUs.

This paper makes the following contributions:

- We present the *most efficient* blocking mechanism that reduces memory bandwidth usage and reduces overhead due to blocking. Our SC and LBM implementation is no longer memory bandwidth bound even for very large grids.
- By making SC and LBM compute bound, our implementation effectively utilizes the available thread- and data-level parallelism on modern processors. We scale near-linearly with multiple cores and SIMD width on both CPUs and GPUs.
- We present a *flexible load-balancing scheme* that distributes the grid elements equally amongst the available threads, all of which perform the same amount of external memory read/write and stencil computations.
- We obtain the fastest implementation of 7-point stencil and LBM on a single-socket Intel Core i7 CPU for both single- and double-precision inputs. Our 7-point stencil and LBM implementations are 1.5X and 2.1X, respectively faster than the best reported numbers. As long as the cache capacity is large enough to hold the blocked data, our 3.5D blocking achieves close to peak

compute throughput for both CPUs and GPUs.

- On Nvidia GTX 285 GPU, our 7-point stencil for single-precision inputs is 1.8X faster than previously reported numbers. Our GPU implementation employs an efficient work distribution across thread warps, thereby reducing the overhead of parallelization.

II. RELATED WORK

Stencil computations (SC) are challenging for state-of-the-art multi-core architectures because of high memory bandwidth requirements. Data reuse is small and grid sizes are usually larger than caches, forcing subsequent sweeps through the grids to reload data. Spatial blocking [10], [11], [12], [13], [14], [15], [16], [17], [18] has been a commonly used approach to improve cache locality and expose parallelism.

Datta et al. [10], [11] propose an auto-tuning approach for SC to select appropriate blocking parameters for several architectures. Their implementation is memory bound when running 4 CPU cores or more than 1 IBM Cell socket. Their implementation on Nvidia GTX280 is compute bound for double-precision results (double-precision throughput is low) but is memory bound for single precision. Similarly, Williams et al. [18] use an auto-tuning approach to optimize an LBM variant, called LBMHD for various multi-core architectures.

Blocking in the spatial dimensions only provides limited reuse opportunities because each grid point is used only a handful of times. We find that spatial blocking is in general insufficient to turn bandwidth-bound stencils (for instance, single-precision 7-point stencil or LBM) into compute-bound algorithms on modern CPU and GPU architectures.

There has been previous work in using temporal blocking for SC and LBM [13], [17], [19], [20], [21], [22], [23]. Habich et al. [13] block across different time steps t for LBM to reuse grid data within cache before writing back to memory. However, they do not perform spatial blocking and their scheme does not extend to larger grids where XY slabs do not fit in the last level cache. Williams et al. [21] performs 4D blocking of SC on IBM Cell architecture and parallelizes across 4D trapezoids. In contrast, we propose a 3.5D blocking scheme for less overhead and apply it to both SC and LBM for CPU and GPU architectures. Moreover, we perform fine-grain parallelization within an XY slab.

Parallelizing temporal blocking must be done carefully since the blocking can introduce overlapping memory accesses and computation that must be carefully tuned to minimize overheads. The thread scheduling algorithm used in Habich et al. for LBM [13] and Wellein et al. for 7-point stencils [17] exposes parallelism by assigning temporal wavefronts to several threads. Their scheduling scheme has two limitations: (1) it limits the temporal blocking factor dim_T to the number of cores, limiting bandwidth gains to that number, and (2) having a subset of cores accessing memory while other cores operating within cached buffers introduces imbalance in bandwidth resource utilization, reducing effective memory bandwidth. Our scheduling is more flexible as we partition each XY slice across all threads, each of which reads from memory for time

step t, uses cached buffers for the intermediate time steps, and writes to memory for $t + dim_T - 1$. Furthermore, we can choose dim_T to target the necessary bandwidth reduction, independent of the number of cores available.

Recently, there are a number of publications that map SC and LBM to graphics processing units (GPU) [15], [24], [25], [26], [27], [28], [29]. Because SC exhibits localized computation, nearest neighbor interaction, and streaming data accesses, it maps well to GPUs with many cores, non-coherent fast memory, and high memory bandwidth. Current Nvidia GPU has a small 16KB shared memory at each multiprocessor, making tiling while keeping overhead low challenging. To allow concurrent processing of multiple blocks, grid cells at block boundaries must be replicated. To keep overhead low, the blocks should be sufficiently large such that the boundary cells are much fewer than the non-boundary ones.

Many publications compare highly optimized GPU implementations to CPU implementations that have not been fully optimized to effectively exploit on-chip caches to reduce memory bandwidth requirement. For example, when LBM is bandwidth limited, it benefits very little from on-chip parallelism such as SIMD vector and multiple cores. For instance, Bailey et al. [24] use CUDA on GTX8800 to deliver 28X better in performance than their 4-core CPU implementation at 8.99 million lattice updates/second (MLUPS). This CPU implementation does not scale with the number of cores and does not use SIMD.

III. MODERN ARCHITECTURES

For best performance of stencil computations, external memory bandwidth should be judiciously utilized while exploiting instruction-level parallelism (ILP), thread-level parallelism (TLP), and data-level parallelism (DLP).

A. Memory Latency

Memory instructions must go through a virtual-to-physical address translation, which is in the critical path of program execution. To improve translation speed, a translation look aside buffer (TLB) is used to cache virtual-to-physical translation of most frequently accessed pages. If the translation is not found in the TLB, processor pipeline stalls until the TLB miss is served. Both last-level cache (LLC) and TLB misses are difficult to hide because the miss penalty reaches more than a few hundred cycles. The LBM kernel suffers from both high TLB misses and LLC misses because of its streaming access patterns. At each time step, the entire lattice is brought into cache only to be evicted before any reuse. Thus, all accesses to the lattice structure always suffer external memory latency and utilize external memory bandwidth. The large number of streams can result in filled up read and write buffers and can cause TLB conflicts. In later sections, we describe in detail how to improve cache reuse and reduce external memory bandwidth usage. The 7-point stencil involves reduction operations that introduce dependencies in executing a single stencil computation. Hence we require software techniques such as

Platform	Peak BW	Peak Gops		Bytes/Op	
		SP	DP	SP	DP
Core i7	30	102	51	0.29	0.59
GTX 285	159	1116	93	0.14	1.7

TABLE I

PEAK BANDWIDTH (GB/SEC), PEAK COMPUTE IN SINGLE AND DOUBLE PRECISION (GOPS), AND BYTES/OP OF CORE i7 AND GTX 285.

software pipelining and loop unrolling to increase the amount of data-level parallelism.

B. TLP/DLP

Once memory latency impact is minimized, we can exploit high density compute resources in modern processors, which have integrated many cores, each with wider SIMD units. Parallelization and SIMDification are two key techniques to exploit compute resources. Parallelizing stencil computations must be done carefully since the blocking can introduce overlapping memory accesses and computation that must be carefully tuned to minimize overheads. The replicated grid sites are called ghost sites and must be kept small relative to the block size to amortize the overhead of loading extra data. This puts a limit on how small the block size can be and, consequently, on the minimum size of on-chip caches or fast storage. Parallelization overhead through synchronizations can be significant even though all cores sit on the same chip. To this end, we implement our own barrier that is 50X faster than pthreads barrier. As for exploiting SIMD, we can use SIMD units to process multiple stencil or lattice sites concurrently. For LBM, the neighboring velocity vectors must be stored in structure-of-arrays format to enable SIMD processing without gathering data from disjoint regions of memory.

C. Memory Bandwidth

With the increased compute capability, the demand for data also increases proportionally. However, main memory bandwidth is growing at a lower rate than compute [30]. Therefore, we will not be able to scale up to the increasing number of cores and wider SIMD if applications become bandwidth bound. To bridge the enormous gap between processor bandwidth requirements and what the memory system can provide, most processors are equipped with several levels of on-chip memory storage (e.g., caches on CPUs and shared buffer on GPUs). If data structures being accessed fit in this storage, no external bandwidth is utilized, thus amplifying the effective memory bandwidth. However, if data structures are too big to fit in caches, we should ensure that a cache line brought from the memory be fully utilized before evicted out of caches (called cache line blocking). Reorganizing data structure from array-of-structures to structure-of-arrays basically rearranges data elements so that the subsequent elements to be used also reside within the same cache line. Lastly, for SC, spatial and temporal blocking of grids improves data reuse and reduces effective memory bandwidth requirements. Blocking keeps SC kernels compute bound and allows them to scale with more cores and wider SIMD vectors.

D. System Environment

The Intel Core i7 CPU is the latest multi-threaded multi-core Intel-Architecture processor. It offers *four* cores on the same die running at a frequency of 3.2GHz. The Core i7 processor cores feature an out-of-order super-scalar micro-architecture, with newly added 2-way hyper-threading. In addition to scalar units, it also has 4-wide SIMD units that support a wide range of SIMD instructions [31]. Each core has an individual 32KB L1 cache and a 256KB L2 cache. All four cores share an 8MB last-level cache (LLC). The Core i7 processor also features an on-die memory controller that connects to three channels of DDR memory. We measured the performance of our kernels on the Core i7 running SUSE Enterprise Server 11 with 6GB of DDR3 memory overclocked to 1333MHz.

The GTX 285 is a recent GPU from NVIDIA. It has a SIMT architecture with 30 streaming multiprocessors (SMs) running at 1.55 GHz. Each SM has 8 scalar compute units for single-precision and 1 double-precision unit. Each unit is capable of performing a multiply-add (madd) op. In addition, it has a special function unit (SFU) typically used for transcendental operations in single precision.

E. Bytes-Op Ratio

We now look at the peak compute capabilities and available memory bandwidth of modern processors and discuss whether compute and bandwidth resources will limit performance. Table I shows the peak bandwidth and compute of the quad-core Intel Core i7 processor running at 3.2 GHz, and the recent GTX 285 GPU architecture from NVIDIA. In the table, $Bytes/Op$ is the ratio of peak BW and peak compute ops. Here, 1 op implies 1 operation or 1 executed instruction, including arithmetic (floating point and integer) and memory instructions. The ops available on the GTX 285 assumes full use of Special Function Units and madd units - typical stencil operations do not utilize such ops (expect for a small number of madd ops) - hence only get a third of the peak SP compute and half of peak DP ops. This makes the actual bytes/op about 0.43 for SP and 3.44 for DP. Note that achievable bandwidths are usually about 20-25% off from peak (we have measured 22 GB/s on Core i7 and 131 GB/s on GTX 285). In the next section, we will compare the bytes/op delivered by the architectures with the bytes/op demand of SC kernels to discuss the limit in performance.

IV. STENCIL KERNELS

Stencil computation (SC) performs nearest neighbor computation on a spatial grid. SC sweeps through the entire grid multiple times, called time steps, updating each grid point with calculations based on its nearest neighbors. We focus on 3D stencils in this paper.

There are various flavors of iterative sweeps of stencil computation. The most commonly used is the *Jacobi type*, which uses two grids, one designated for reads while the other one is designated for writes in the current time step. For the next time step, the roles of the grids are swapped, and the grid that was written to is now read from. Although Jacobi
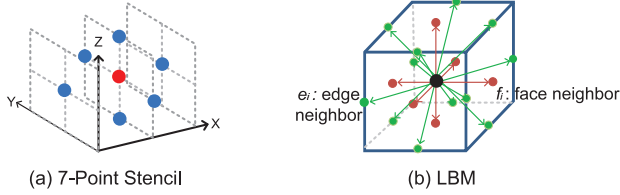
Fig. 1. (a) 7-point Stencil (b) D3Q19 LBM

sweep is the most commonly used, the current implementation on modern processors is bound by the available memory bandwidth, and are the most challenging to scale and fully utilize the increasing computational resources.

We first look at specific examples of stencils and analyze their compute and bandwidth requirements. We discuss the current best implementations of these kernels in light of the above analysis. Many current implementations are bound by memory bandwidth; this motivates the need for blocking in spatial domain and across time steps (or temporal blocking), that we describe in the next section.

A. Partial Difference Equation Solvers

Partial Difference Equation (PDE) solvers are used in many fields such as diffusion and electro-magnetics. Such solvers use stencils that have data reuse in both spatial and time dimensions [10].

1) Seven-Point Stencil: We first consider the 7-point stencil, shown in Figure 1(a), because it is simple yet exhibits the behavior we try to address. A 7-point stencil operation can be represented by the following equation:

$$B_{x,y,z}(t+1) = \alpha A_{x,y,z}(t) + \beta(A_{x-1,y,z}(t) + A_{x+1,y,z}(t) +$$
$$A_{x,y-1,z}(t) + A_{x,y+1,z}(t) + A_{x,y,z-1}(t) + A_{x,y,z+1}(t))$$

A destination grid $B_{x,y,z}$ for time step $t+1$ is updated with a stencil that fetches a grid point from 7 points of a source array A for time step t: $A_{x,y,z}$ and its 6 neighbors in x-1, x+1, y-1, y+1, z-1, z+1 directions. The grid point is scaled by α and its neighbors are summed and scaled by β and both expressions are combined. The update for each grid point involves 16 ops, of which there are 2 floating point multiplications, 6 additions, 7 reads (loads) from A and 1 write (store) to B. There are a total of 8 memory values accessed, which correspond to 32 bytes in single precision (SP) and 64 bytes in double precision (DP). However, there is significant reuse of data from A across spatial neighbors. With spatial blocking, a chunk of data from A can be loaded into local memory of different architectures and reused for different computations. Each interior point (not at the boundary) in the block is reused as many times as the stencil size (7 in this case). There is thus only 1 value read from A from external memory and 1 value written to B, for a total of 8 bytes for SP and 16 bytes for DP. The *bytes/op* for 7 point stencil is thus 0.5 for SP and 1.0 for DP.

There are two potential sources of extra data traffic. For cache-coherence architectures like the Intel Core i7, each write to memory first involves a fetch of the cache line from memory and subsequent write-back traffic. This extra data transfer can

be eliminated using streaming stores to memory. However, there is also additional traffic since two *ghost layers* of points in each blocked dimension (for the plus and minus 1 points in each dimension) needs to be brought in for computing stencils on the boundary points. These ghost layers are only loaded but not computed on. This results in extra data traffic, hereafter called *overestimation*. The extent of the extra data traffic per grid point depends on the dimensionality of the blocking, as well as the block size. Hence, we advocate using a 2.5D spatial blocking scheme in Section V-A3.

2) Twenty-seven point stencil: For the 27-point stencil, each grid point computation involves the edge, face and corner points of a 3x3x3 cube surrounding the center grid point. The edge points, corners and face neighbors are multiplied by different constants. The number of ops is now 58: 4 multiplies, 26 adds, 27 loads from A and 1 store to B. After spatial blocking, we still only read and write one element per stencil computation. *Bytes/op* is thus 0.14 for SP and 0.28 for DP.

B. Lattice Methods

Lattice methods work on more than one value per grid point. Shown in Figure 1(b), we use the 3-dimensional D3Q19 LBM that works on 19 values per grid point, and produces a new set of 19 values. As opposed to the 7- and 27-point stencils in the previous section, there is no reuse of data among different grid points, thus making it heavily memory bound.

In each time step, LBM performs the following actions on the entire lattice:

1. Read 19 values from the current cell of a source array.
2. Calculate new values
3. Propagate the 19 new values to 18 neighboring sites and the local site by updating a destination array.
4. Swap the source and destination array. That is, the destination array of time step t will be the source array of the time step $t + dt$.

Each cell update requires reading 19 values plus a flag array to find if the cell is an obstacle or boundary and writing 19 values. The number of ops required for each grid cell update is 259: 220 flops (about 12 flops per direction) plus 20 reads plus 19 writes. For SP data, the number of bytes read is 76-80 bytes (depending on how the flag is stored), while 76 bytes are written. For efficient use of SIMD on different architectures, each of the 19 values per cell are stored in different arrays (Structure-of-Arrays configuration) rather than all together in a single array (Array-of-Structures). However, this results in writes to some neighboring values that are not aligned to cache-line boundaries (every cell i will write to positions $i+1$, $i-1$, etc.). Streaming stores are generally not possible in such cases. Note this does not happen in normal stencils since such operations only write their own values and not neighbors. In the absence of streaming stores, the number of bytes written is 152, for a total of about 228 bytes in SP (and 456 bytes in DP). *Bytes/op* for LBM is thus 0.88 for SP and 1.75 for DP.

C. Current Best Implementations of Stencils

Different architectures offer different bandwidth-to-compute ratios, which can differ widely between SP and DP. Given a particular application, if the bytes/op of the application is higher than the bytes/op offered by the architecture, then the application will be bound by memory bandwidth and any optimization to improve compute is ineffective until bandwidth requirement is decreased.

LBM has high bytes/op of 0.88 for SP and 1.75 for DP. This makes the SP version bandwidth-bound on both CPU and GPU architectures (see Bytes/Op in Table I). The DP version is bandwidth-bound on CPU but is compute-bound on GPU. The 7-point stencil has bytes/op of 0.5 for SP and 1.0 for DP. SP is bandwidth-bound on both CPU and GPU. DP is bandwidth-bound on CPU and compute-bound on GPU. The 27-point stencil has low bytes/op that is sufficient to make it compute bound on both architectures.

While spatial blocking techniques are sufficient to make 27-point stencil compute bound, the bytes/op of the 7-point and LBM kernels are much higher than the peak bytes/op available on the architecture. In order to further decrease the bytes/op, we advocate temporal blocking in conjunction with appropriate spatial blocking to fully utilize the architectural compute resources.

V. 3.5D BLOCKING (2.5D SPACE + 1D TIME)

As explained in Section IV, we focus on improving the throughput of stencil-based kernels for which the bandwidth-to-compute ratio (γ) is greater than the corresponding peak bandwidth to peak compute ratio (Γ) of the underlying architecture. The performance of such kernels is typically bound by the available memory bandwidth, and as a result, these kernels *do not* completely utilize the available compute power. Although we focus on CPU and GPU, the formulation developed and described in this section is applicable to a broad range of computing devices such as Intel Larrabee [8], etc.

Notation: Let $\mathcal{R}$ denote the *radius* (in units) of extent of the stencil, usually defined as the Manhattan distance (e.g. k-point stencil) or L_∞ norm (e.g. LBM)). Let $\mathcal{C}$ denote the size of the available *fast* on-chip storage (LLC and registers for CPU, shared memory and registers for GPU). We assume the 3D data is laid out with the X-axis being the most frequently varying dimension, followed by the Y- and Z-directions. Without loss of generality, let (0,0,0) (origin) denote the grid point with minimum coordinates, and $\mathcal{N}_X$, $\mathcal{N}_Y$ and $\mathcal{N}_Z$ denote the input grid dimensions in the X, Y and Z directions respectively. Let $\mathcal{P}$ denote a grid point, with $|\mathcal{P}|$ denoting the distance of the grid point from the origin. The size of each grid element is denoted by $\mathcal{E}$.

A. Spatial Blocking

We describe three techniques aimed at exploiting all data reuse available in the stencil operation.
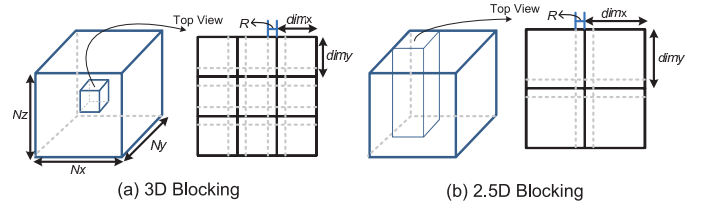

Fig. 2.   (a) 3D blocking (b) 2.5D blocking

1) Wavefront Blocking: Spatial Blocking techniques are aimed at *completely utilizing* each data element that is loaded into the on-chip storage. For stencil-based kernels, this amounts to using a wave-front based computation pattern in steps, starting from the origin and moving diagonally away. At every step (s), the amount of data required to be resident in the on-chip memory is the set of grid points ($\mathcal{P}_s$) such that $(s - \mathcal{R}) \leq |\mathcal{P}_s| \leq (s + \mathcal{R})$.

This methodology suffers from two performance concerns. First, the working set increases as the wave-front approaches the center of the grid ($\mathcal{O}(\mathcal{N}_X^2 + \mathcal{N}_Y^2 + \mathcal{N}_Z^2)$ grid elements). For practical input sizes, this far exceeds the available on-chip memory, thereby requiring multiple loads of grid elements, and increase in corresponding memory bandwidth. Second, thread-level parallelism is extracted by evenly distributing the grid points amongst the threads. Because of the irregular shape of the working set, this distribution involves computation overhead and inter-thread communication of grid elements. As far as DLP is concerned, it requires gather operations since the elements are not laid out in a contiguous fashion.

2) 3D Blocking: In contrast, a commonly used technique is to divide the input into overlapping axis-aligned 3D blocks (say of dimension $dim_X^{3D}, dim_Y^{3D}, dim_Z^{3D}$) (Figure 2(a)). To perform the computation, each block is loaded into the on-chip memory, and kernel computation is performed on grid elements that have all of their required stencil elements within the boundaries of the loaded block. Since the grid elements within distance $\mathcal{R}$ of the boundary (termed as *ghost layer*) do not satisfy this constraint, the computation is only performed for $(dim_X^{3D}-2\mathcal{R})x(dim_Y^{3D}-2\mathcal{R})x(dim_Z^{3D}-2\mathcal{R})$ elements within a block. Since the complete block needs to reside in the on-chip memory, this requires $(\mathcal{E} \, dim_X^{3D} \, dim_Y^{3D} \, dim_Z^{3D}) \leq \mathcal{C}$. As far as the bandwidth is concerned, the elements in the ghost layer are loaded multiple times. The ratio of extra bandwidth required (κ^{3D}) is around [1] $((1 - 2\mathcal{R}/dim_X^{3D})(1 - 2\mathcal{R}/dim_Y^{3D})(1 - 2\mathcal{R}/dim_Z^{3D}))^{-1}$. The total amount of floating-point operations performed is *still the same*, although the number of loads increases by the same factor κ^{3D}.

In order to *minimize* the amount of extra bandwidth, dim_X^{3D}, dim_Y^{3D} and dim_Z^{3D} are set equal to each other, with each being set to $\lfloor \sqrt[3]{\mathcal{C}/\mathcal{E}} \rfloor$. As evident from the formula, κ^{3D} is inversely proportional to $\mathcal{R}/dim_X^{3D}$. For example, with $\mathcal{R} \sim 10\%$ of dim_X^{3D}, κ^{3D} is around 1.95X, and for $\mathcal{R} \sim 20\%$ of dim_X^{3D}, κ^{3D} increases to 4.62X, a huge overestimation.

---

[1]Since memory transfers are executed at the granularity of cache lines, the actual amount of overestimation may be larger – depending on the number of partial cache lines loaded.

As far as the stencil computation is concerned, the computation follows a specific pattern. Say we perform computation for all the grid elements within the XY plane for a specific Z value (initialized to 0), and increment Z until all grid elements have been operated upon. For any specific Z value (Z_s), we only require grid elements within (Z_s+/-$\mathcal{R}$) Z planes to be resident in the cache, a total of $2\mathcal{R}+1$ planes. In theory, this can be much smaller than dim_Z^{3D}. This allows for blocks with much larger dim_X and dim_Y to be cache resident, leading to reduction in extra bandwidth required and executed ops as described next.

3) 2.5D Spatial Blocking: We exploit the fact that only ($2\mathcal{R}+1$) XY planes need to be cache resident. We therefore perform blocking in 2D (XY) plane, and stream through the third dimension (Z). We denote this as 2.5D blocking, borrowing the term 2.5D from field of computer graphics [2] (Figure 2(b)). Let $dim_X^{2.5D}$ and $dim_Y^{2.5D}$ denote the blocking dimensions in X and Y direction respectively. We denote the region within each plane as an XY sub-plane. Since the buffer needs to completely reside in the cache, ($\mathcal{E}(2\mathcal{R}+1)dim_X^{2.5D}dim_Y^{2.5D}) \leq \mathcal{C}$. Before describing the formulation for evaluating the blocking dimensions, we first describe the complete 2.5D blocking algorithm.

We denote the blocking data structure as $\texttt{Buffer}^{2.5D}$, with dimensions ($dim_X^{2.5D}, dim_Y^{2.5D}, (2\mathcal{R}+1)$). $\texttt{Buffer}^{2.5D}$ should have enough space for ($2\mathcal{R}+1$) XY sub-planes, with the ith sub-plane in the buffer denoted as $\texttt{Buffer}^{2.5D}[i]$. For each XY sub-plane, the computation consists of the following 2 phases:

Phase 1: *Prolog* – Load the grid elements for the XY sub-plane with $z \leftarrow [0 \mathbin{..} 2\mathcal{R})$ into $\texttt{Buffer}^{2.5D}[0 \mathbin{..} 2\mathcal{R})$ respectively.

Phase 2: *Cache-Friendly Stencil Computation* – For each $z \in [\mathcal{R} \mathbin{..} \mathcal{N}_Z)$
(a) Load the XY sub-plane for (z+$\mathcal{R}$) into $\texttt{Buffer}^{2.5D}[(z+\mathcal{R})\%(2\mathcal{R}+1)]$.
(b) Perform Stencil Computation for XY sub-plane stored in $\texttt{Buffer}^{2.5D}[z\%(2\mathcal{R}+1)]$ and store the result into external memory.

Note that unlike 3D blocking, there is no extra bandwidth requirement in the Z direction as we stream through it. In addition, the access pattern from external memory is very regular and can be captured by various hardware pre-fetchers. As far as the extra bandwidth($\kappa^{2.5D}$) for the ghost layer is concerned, it is equal to $((1-2\mathcal{R}/dim_X^{2.5D})(1-2\mathcal{R}/dim_Y^{2.5D}))^{-1}$. Minimizing the overestimation yields $dim_X^{2.5D}(=dim_Y^{2.5D}) = \lfloor \sqrt[2]{\mathcal{C}/(\mathcal{E}(2\mathcal{R}+1))} \rfloor$. Citing similar examples as before, for $\mathcal{R} \sim 10\%$ of dim_X^{3D} (i.e., using the same $\mathcal{R}$ value from the 3D-blocking example), $\kappa^{2.5D}$ is around 1.2X and $\mathcal{R} \sim 20\%$ of dim_X^{3D}, $\kappa^{2.5D}$ increases to only 1.77X, around 2.6X reduction over 3D blocking.

---

²2.5D refers to objects emanating from the ground (Z = 0) and growing in the Z direction.

In summary, 2.5D blocking helps reduce the extra bandwidth (as compared to loading each element once in a small fixed size on-chip memory) and the corresponding extra computation required to perform stencil computation. This may lead to substantial speedups in the runtime, especially when used in conjunction with the temporal blocking scheme we next describe.

B. 1D Temporal Blocking

Although 2.5D spatial blocking ensures near-optimal usage of memory bandwidth, it still does not guarantee full utilization of the computational resources. Since the original application executes multiple time steps (usually hundreds to thousands), the only way to reduce bandwidth is to execute several time steps of the blocked data so that the intermediate data can reside in the cache, and then store the resultant output to the main memory. This proportionately reduces the amount of bandwidth.

Although temporal blocking has been used in the past [12], [13], we provide: (a) algorithm for combining temporal blocking with 2.5D spatial blocking to achieve the compute and bandwidth friendly 3.5D blocking, (b) thread-level and data-level parallel algorithms to fully exploit the available compute resources.

For temporal blocking, let us denote dim_T as the number of time steps we need to block before writing the data to the external memory. To accommodate for dim_T time steps, we extend the buffer to store the XY sub-planes for dim_T time instances (starting from 0 to (dim_T-1)). After the (dim_T-1) time steps, the output for the next step is written out to the external memory. We describe later the number of XY sub-planes required for each time instance.

We now describe the overall computation flow of the 3.5D blocking algorithm followed by the detailed algorithm and formulation of the various parameters.

C. 3.5D Computation Flow

We advocate the use of 2.5D spatial blocking rather than 3D spatial blocking for stencil computations since it yields lower bandwidth and compute overestimation. Our 3.5D blocking scheme is a combination of the 2.5D spatial blocking scheme and the 1D temporal blocking scheme. For ease of explanation, say $\mathcal{R}$ equals 1. The (z = i) plane is referred to as z_i. Also z_0 (boundary condition) does not change with time. Let dim_T equal to 3. Consider Figure 3(a). Let $\mathcal{S}_i$ denote the ith step of the execution. $\mathcal{S}_i$ at the time instance t' computes the grid elements for a specific z_s value by reading grid elements from [z_s-1, z_s, z_s+1] at time instance t'-1. For example, $\mathcal{S}_9$ computes grid elements for $z_3(t' = 1)$, and $\mathcal{S}_{21}$ computes grid elements for $z_2(t' = 3)$.

Furthermore, note that only the steps executed for $t' = 0$ and $t' = 3$ ($=dim_T$) read and write data from/to the external memory, respectively. All the intermediate steps (in this case $t' = 1$ and $t' = 2$) have no communication with the external memory. The resultant bandwidth reduces by a factor of 3
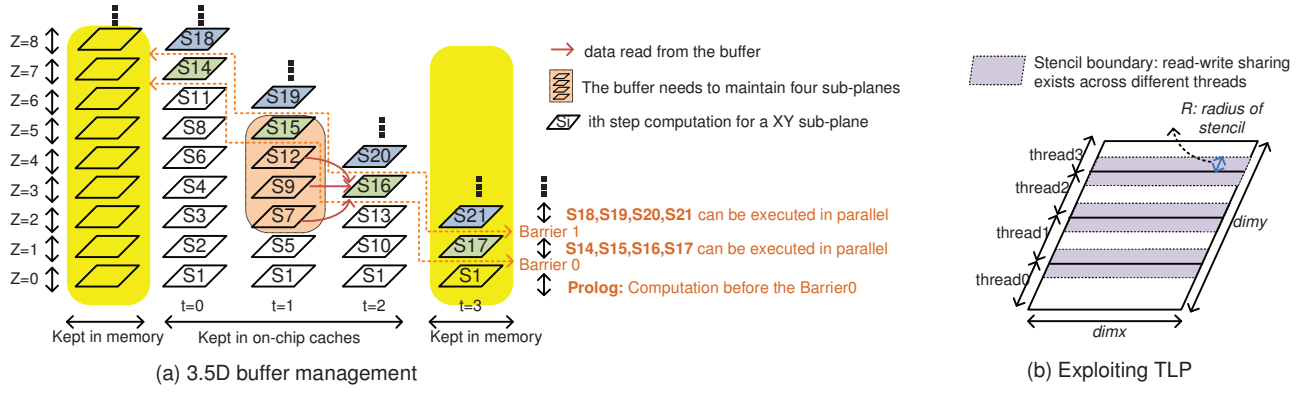
Fig. 3. (a) 3.5D Buffer management (for $\mathcal{R}$=1, dim_T=3) (b) Exploiting thread-level parallelism for stencil computation

(dim_T in general). We now determine the number of XY sub-planes needed to be stored for each time instance. Consider a step (say $\mathcal{S}_{16}$, at $t' = 2$). This requires $\mathcal{S}_7$, $\mathcal{S}_9$ and $\mathcal{S}_{12}$ to be completely executed. Thus we require at least $3(=(2\mathcal{R}+1))$ sub-planes for the time instances $[0 .. (dim_T - 1)]$. However, note that this requires one time step to finish completely before executing the step at the next time instance. This scheme has two potential performance delimiters on current multi-/many-core platforms:

(1) *Barrier Synchronization* between various cores after each step.

(2) *Limited amount of parallelism:* Since there is a barrier after each step, the only parallelism available is within the stencil computation during a step. Although this may be enough for a few cores, this scheme would not scale to the increasing number of computational cores.

In order to scale with larger number of computational cores, we propose the following extension. Instead of storing 3 sub-planes per time instance, we can store one more (= 4) sub-planes. This ensures that one step at each time instance can be executed in parallel with each other, including reading and writing data from/to different buffers. Consider steps $\mathcal{S}_{18}$, $\mathcal{S}_{19}$, $\mathcal{S}_{20}$ and $\mathcal{S}_{21}$. While $\mathcal{S}_{18}$ is updating the buffer, $\mathcal{S}_{19}$ reads from data stored by $\mathcal{S}_8$, $\mathcal{S}_{11}$ and $\mathcal{S}_{14}$. Similarly $\mathcal{S}_{20}$ reads from data stored by $\mathcal{S}_9$, $\mathcal{S}_{12}$ and $\mathcal{S}_{15}$ and finally $\mathcal{S}_{21}$ reads from data stored by $\mathcal{S}_{10}$, $\mathcal{S}_{13}$ and $\mathcal{S}_{16}$. By storing 4 XY sub-planes per time instance, these steps can be executed in parallel. This increases the total amount of available parallelism by a factor of dim_T, and also reduces the overhead of barrier synchronization. Hence for a general scenarios, we need to maintain $(dim_T) \times (2\mathcal{R}+2)$ XY sub-planes in the caches.

As far as the size of each XY sub-plane is concerned, since all the buffers need to reside in the cache,

$$(\mathcal{E}(2\mathcal{R}+2)dim_T dim_X^{3.5D} dim_Y^{3.5D}) \leq \mathcal{C}. \tag{1}$$

Let us now compute the extra bandwidth required. Consider the ghost layer. After one time step, the grid elements within distance $\mathcal{R}$ of the boundary do not have the updated values. Since we perform dim_T time steps, only the data within $(dim_X^{3.5D}-2\mathcal{R}dim_T)x(dim_Y^{3.5D}-2\mathcal{R}dim_T)$ XY sub-plane is correct, and needs to be written out. Hence the ratio

of extra bandwidth, or overestimation, $(\kappa^{3.5D})$ is given by

$$\kappa^{3.5D} = ((1-2\mathcal{R}\,dim_T/dim_X^{3.5D})(1-2\mathcal{R}\,dim_T/dim_Y^{3.5D}))^{-1} \tag{2}$$

In order to maximize the use of computational resources, dim_T should be greater than the ratios of the bandwidth-to-compute of the kernel (γ) to the peak bandwidth-to-compute of the machine (Γ). Hence

$$dim_T \geq \eta(= \lceil \frac{\gamma}{\Gamma} \rceil) \tag{3}$$

Minimizing the overestimation yields

$$dim_X^{3.5D} = dim_Y^{3.5D} = \lfloor \sqrt[2]{\mathcal{C}/(\mathcal{E}(2\mathcal{R}+2)(\eta))} \rfloor. \tag{4}$$

In addition to extra bandwidth, temporal blocking also introduces the overhead of extra computation that needs to be performed repeatedly for the grid elements within the ghost layers. The ratio of extra computation is similar to $\kappa^{3.5D}$.

D. Exploiting Thread and Data Level Parallelism

Exploiting Thread-Level Parallelism: Let $\mathcal{T}$ denote the total number of threads. As explained in Section V-C, the buffers are allocated in a way that we can perform independent computation on each time instance. Therefore, there are *two* different ways to divide computation amongst the available threads:

(1) Assign each time instance to a thread. This requires a barrier after each thread is done.

(2) Divide each XY sub-plane amongst the $\mathcal{T}$ threads. There needs to be a barrier after each round of dim_T XY sub-plane computation.

Since dim_T may be much smaller than $\mathcal{T}$, we advocate and have implemented (2), which scales well with increasing number of cores. We divide $dim_Y^{3.5D}$ by the number of threads, and assign each thread the relevant rows (Figure 3(b)). In case $dim_Y^{3.5D} < \mathcal{T}$, each thread gets partial rows for each XY sub-plane. Note that this technique of dividing work also ensures that each thread reads/writes the same amount of data from/to external memory and also executes similar number of ops – thereby lending itself well to parallel implementation.

Exploiting Data-Level Parallelism: Since our 3.5D blocking makes the stencil computation compute bound, we can further reduce the executed instructions by taking advantage of the SIMD execution units available on the modern computing units – including both CPU and GPU. We exploit SIMD by performing stencil operation on multiple grid elements together. Since the grid elements are stored in a contiguous fashion, this usually involves vector load/store operations, although some of them may be from unaligned addresses, that may reduce the efficiency of execution. We describe the specific implementation details for the two architectures in Section VI.

E. Parallel 3.5D Blocking Algorithm

We now describe the parallelized 3.5D blocking algorithm. Given the cache size ($\mathcal{C}$), and the element size ($\mathcal{E}$), the various blocking parameters ($dim_X^{3.5D}$, $dim_Y^{3.5D}$ and dim_T) are computed. The XY plane is divided into overlapping XY sub-planes of dimension $dim_X^{3.5D}$ X $dim_Y^{3.5D}$. In addition, each thread is pre-assigned a specific part of each XY sub-plane. For each XY sub-plane, the computation proceeds in the following 3 phases:

Phase 1: *Prolog* – Load and perform all the stencil computation required before performing $z = \mathcal{R}$ for $t' = dim_T$. In the example illustrated in Figure 3, this consists of performing steps $\mathcal{S}_1 .. \mathcal{S}_{13}$ in chronological order.

Phase 2: *Stencil Computation* – For each $z \in [\mathcal{R} .. (\mathcal{N}_Z - 2dim_T\mathcal{R}))$, perform the memory read ($t' = 0$) and stencil computation for $t' = [1 .. dim_T]$. The computation performed for $t' = dim_T$ is written to external memory, while $t' = [1 .. dim_T)$ write to the relevant buffer addresses. All the threads simultaneously work on their assigned regions in the XY sub-planes for all the time instances. There is a barrier after each thread has finished its computation before moving to the next z. The specific z_s at $t' = t''$ is equal to $z + 2\mathcal{R}(dim_T - t'')$. The Buffer index for any z_s equals $z_s \% (2\mathcal{R} + 2)$.

Phase 3: *Epilog* – At the end of Phase 2, all XY sub-planes have been read from external memory, and the stencil output for $t' = dim_T$ for all $z < (\mathcal{N}_Z - 2dim_T\mathcal{R}))$ have been written to external memory. The remaining stencil computations are carried out and result for $t' = dim_T$ for the remaining z values written to external memory.

Throughout the computation, the grid elements are read, written and operated in a SIMD fashion, thereby utilizing all the computational resources. As evident from the above algorithm, as long as all the buffer entries reside in the on-chip memory, performing dim_T time steps only requires grid elements to be read and written once from/to the external memory, resulting in a bytes/op ratio reduction of $dim_T/\kappa^{3.5D}$. As long as this decrease in byte/op ratio ($dim_T/\kappa^{3.5D}$) $> \gamma/\Gamma$, the resultant computation should completely utilize the available computational resources, as opposed to the original bandwidth bound implementation.

VI. IMPLEMENTATION ON MODERN ARCHITECTURES

We now describe the specific details of our 3.5D blocking algorithm for 7-point stencil and LBM on both CPU and GPU. For CPU, the 8MB LLC is available for blocking, while on GPU, the 16KB scratch pad and the 64KB register file is the memory available to perform the 3.5D blocking. For both 7-point stencil and LBM, $\mathcal{R}$ equals 1.

On CPU, we exploit thread-level parallelism using pthreads, and data-level parallelism using SSE [31]. All CPU implementations use large memory pages (2 MB) to minimize TLB misses, which improve performance between 5% and 20%. For GPU, we use the CUDA [32] programming model to exploit both thread and data level parallelism.

A. 7-Point Stencil

CPU Implementation:
Comparing the bandwidth-to-compute ratio the kernel (γ) from Section IV to that (Γ) of Core i7 from Table I, we see that $\gamma > \Gamma$ for both SP ($0.5 > 0.29$) and DP ($1.0 > 0.59$). Thus our 3.5D blocking would benefit both SP and DP. The minimum value of dim_T that satisfies equation 3 is ($dim_T = 2$). Since higher values of dim_T result in increasing overestimation, we chose ($dim_T = 2$) for our implementation. In order to determine $dim_X^{3.5D}$ and $dim_Y^{3.5D}$, we use $\mathcal{C}$ equal to 4MB (half of cache size). This is done since the cache is used to store other data structures used by the application (and other processes), and also the loads from the external memory cannot bypass the cache, thereby reducing the total size available for the blocking data structure.

For SP, $\mathcal{E}$ equals 4 bytes, and hence from equation 1, we get $((4)(4)(2)dim_X^{3.5D} dim_Y^{3.5D}) \leq$ 4MB. To minimize the overestimation factor $\kappa^{3.5D}$, we use equation 4 and get: $dim_X^{3.5D} \leq 361$. We used $dim_X^{3.5D} = dim_Y^{3.5D} = 360$. For **SP**, $\kappa^{3.5D}$ evaluated to around **1.02X**.

The Core i7 has 4 cores. To exploit TLP, each thread was assigned 360/4 = 90 rows in the XY sub-plane. Our methodology for dividing rows between threads also minimizes the inter-cache communication (L1, L2), since only the boundary rows need to read data written by other cores, thereby reducing the inter-core communication. Since the buffer data structure is frequently accessed, we do not expect it to be evicted from caches, therefore the achieved performance should closely match the expected performance. We present performance results in Section VII-A. To exploit DLP, we performed stencil computation on 4 elements together using the single-precision floating point SSE instructions. Depending on the alignment of the memory, we did require unaligned load/store instructions.

For DP, $\mathcal{E}$ equals 8 bytes, which leads to $dim_X^{3.5D} \leq 256$. We used $dim_X^{3.5D} = dim_X^{3.5D} = 256$. For exploiting TLP, each thread was assigned 64 rows in the XY sub-plane. To exploit DLP, we operated on 2 grid elements simultaneously using double precision floating point SSE instructions. For **DP**, $\kappa^{3.5D}$ evaluated to around **1.04X**.

For comparison purposes, using a 3D blocking alone would still result in a bandwidth bound number, and a 4D (3D spatial + temporal) blocking would have resulted in a
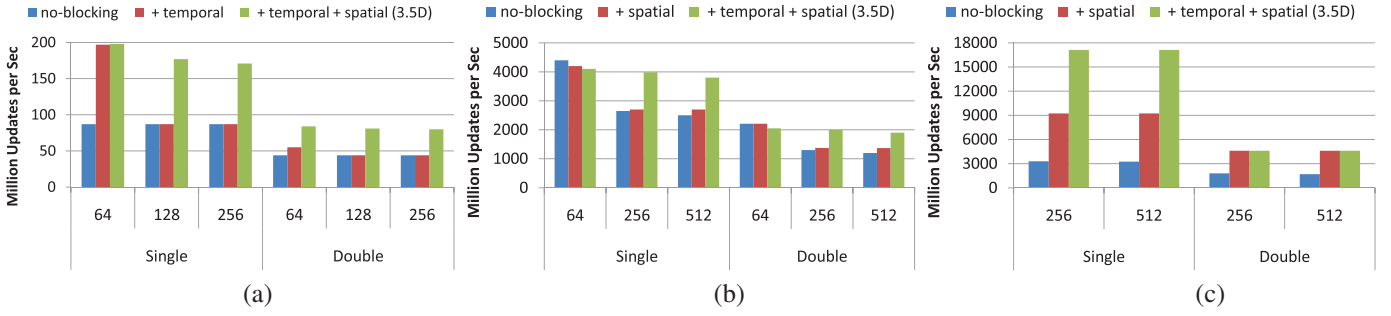
Fig. 4. Performance comparisons of single and double-precision stencils (a) LBM on CPU (b) 7-point stencil on CPU and (c) 7-point stencil on GPU, for different grid sizes with no-blocking, only temporal/spatial blocking and both temporal and spatial blocking (3.5D blocking).

computation overhead of 1.18X for SP, and 1.21X for DP respectively.

GPU Implementation:

For SP, because $\gamma = 0.5$ is greater than $\Gamma = 0.14$, our 3.5D blocking scheme should improve performance. For DP, because $\gamma = 1.0$ is already less than $\Gamma = 1.7$, the original computation should be compute bound and cannot be further sped up. Hence we focus on SP for the rest of this sub-section.

We use the register file (64 KB in total) to store the grid elements (similar to the stencil implementation in Nvidia SDK 3DFD [15]). Furthermore, $\Gamma = 0.14$ assumes full use of special function units, something that stencil computation cannot easily take advantage of. Hence, we use the actual compute flops, and use $dim_T = 2$ for our 3.5D blocking, for which minimizing $\kappa^{3.5D}$ reduces to $dim_X^{3.5D} \leq 45.2$.

The NVIDIA GPU architecture consists of multiple streaming multiprocessors, or SMs, each with multiple scalar processors that execute the same instruction in parallel. In this work, we view them as SIMD lanes. The GTX 285 has eight scalar processors per SM, and hence an 8-element wide SIMD. However, the logical SIMD width of the architecture is 32. Each GPU instruction works on 32 data elements (called a *thread warp*). Global memory (GDDR) accesses on the GPU are optimized for the case that every thread in a warp loads 4/8 bytes of a contiguous region of memory. For 7-point stencils, setting $dim_X^{3.5D}$ to a multiple of 32 enables each thread to load the same amount of data from contiguous memory locations. Since $dim_X^{3.5D}$ should be less than 45.2, it is chosen to be equal to 32, same as the warp size. Since the register file is shared between the threads, each thread maintains a portion of the buffer, namely the grid element corresponding to the specific z_s value. Therefore, each thread stores 4 grid elements per time instance. For performing the stencil computation, each thread needs to read the neighboring values in X and Y direction from the other threads within the block. Since CUDA does not allow for explicit inter-thread communication, we use the shared memory to communicate between threads. Specifically, each thread stores the grid element for the specific z_s value for which the stencil computation is being performed. This is followed by a synchronization between threads. The threads can then read the relevant values from the shared memory and perform the computation. This mechanism for inter-thread communication between threads using the shared memory is employed for all time instances.

For $t' = dim_T$, only the relevant sub-set of the XY sub-plane needs to be written to the external memory – i.e. the rectangular block of $(dim_X^{3.5D} - 2dim_T)X(dim_X^{3.5D} - 2dim_T)$. Hence the threads operating in the ghost layer should not write out their results, which requires overhead of branch instructions and branch divergence between the threads. In order to amortize the overhead of such operations, each thread reads/operates for a few Y values (instead of just one).

The resultant $\kappa^{3.5D}$ evaluates to around **1.31X**.

B. LBM

CPU Implementation:

The bandwidth-to-compute ratio of the kernel (γ) equals 0.88 for SP and 1.75 for DP. Thus our 3.5D blocking would benefit both SP and DP implementations. Using equation 3, $dim_T \geq 2.9$. We chose ($dim_T = 3$) for our implementation.

For SP, $\mathcal{E}$ equals $4*20 = 80$ bytes (19 directions plus a flag array), and hence equation 1 yields $((80)(4)(3)dim_X^{3.5D}dim_Y^{3.5D}) \leq 4MB$. To minimize the overestimation factor ($\kappa^{3.5D}$), $dim_X^{3.5D} \leq 66$. We used $dim_X^{3.5D} = dim_X^{3.5D} = 64$. For **SP**, $\kappa^{3.5D}$ evaluates to around **1.21X**. To exploit TLP, each thread was assigned $64/4 = 16$ rows in the XY sub-plane. This is much higher than the 7-point stencil since each grid element has a size of 80 bytes. Hence the number of grid elements that can be blocked is much smaller, thereby leading to smaller blocking sizes, and larger overheads.

For DP, $\mathcal{E}$ equals $8*20 = 160$ bytes, and hence $dim_X^{3.5D} = dim_X^{3.5D} = 44$. For **DP**, $\kappa^{3.5D}$ evaluates to around **1.34X**. To exploit TLP, each thread was assigned $44/4 = 11$ rows in the XY sub-plane.

For comparison purposes, using a 3D blocking alone would still result in a bandwidth bound number, and a 4D (3D spatial + temporal) blocking would have resulted in a computation overhead of 2.03X for SP, and 2.71X for DP. Such large overheads imply that 4D blocking would only improve the bandwidth bound numbers by a small amount (1.08X for SP, and 1.06X for DP). In comparison, using 3.5D blocking we expect speedups to be 2.2X for SP and 2.0X for DP. In fact, these numbers match the obtained performance results for LBM on CPU (Section VII-B). In order to improve

the performance, we further employed techniques like loop unrolling and software pipelining that gave us a marginal improvement in throughput. Section VII-D has a detailed analysis on the performance improvement due to various algorithmic techniques.

GPU Implementation:

The peak bandwidth-to-compute ratio (Γ) of GTX285 is 0.14 for SP and 1.7 for DP. So we do not expect 3.5D blocking to speedup the DP performance numbers of LBM on GTX285. However, SP should benefit from 3.5D blocking.

For SP, to completely exploit the computing resources, $dim_T \geq 6.1$. Using $\mathcal{C} = 16KB$, and $\mathcal{E} = 160$ bytes, yields $dim_X^{3.5D} \leq 2$, which is too small, since $dim_X^{3.5D}$ needs to be greater than $2\mathcal{R}\,dim_T$ for performing any blocking. Even using the minimum value of $dim_T = 2$, yields $dim_X^{3.5D} \leq 4$, which also does not permit for blocking. We therefore did not implement our 3.5D blocking scheme for LBM on GPU, but believe that with increasing cache sizes on future GPU (e.g. Fermi [9]), $\mathcal{R}\,dim_T$ of two and greater can be implemented to achieve speedups in throughput.

VII. PERFORMANCE EVALUATIONS

This section evaluates the performance of 7-point stencil and LBM on the Core i7 and GTX285 processors using our 3.5D blocking scheme and analyzes the measured results. It is important to note that our GPU numbers do not include the transfer time over the PCIe, since the kernel runs for multiple (hundreds to thousands) time steps. Thus the overhead of data transfer is amortized across these time steps.

A. 7-point stencil

CPU: We obtained significant benefits from both spatial and temporal blocking. Figure 4(b) shows the single and double precision performance on the Core i7 with no blocking, only spatial blocking, and the combined blocking. We use three data sets: a small 64^3 grid, a medium 256^3 grid and a large 512^3 grid. On the small example, the entire data set fits in cache, and blocking does not improve performance. In fact, there are overheads involved in block addressing that lead to slight slowdowns. For the medium and large examples, spatial blocking in itself did not obtain much benefit over no-blocking on cache-based architectures. This is because 3 XY slabs of data (our largest slab of 512^2 DP data is 2 MB) fit well in the 8 MB L3 cache even without explicit blocking. Both numbers are bandwidth bound (achieving about 21 GB/s, close to maximum achievable bandwidth).

The addition of 3.5D blocking converted the bandwidth bound kernel to a compute bound one, resulting in a performance that is only 15% off the performance for small inputs. The difference is mainly due to the overestimation of compute due to ghost cells. This gives a performance of 3,900 million updates/second, a 1.5X speed up over no-blocking, and 1.4X over spatial blocking only. The speedup results are similar for DP data - since both compute and bandwidth scale by a factor of 2, DP performance is

half of the SP performance. Our resultant implementation **scales near-linearly** with the available cores, achieving a parallel scalability of around 3.6X on 4-cores. As explained in Section V-E, we require a barrier after writing each XY sub-plane to the external memory. We implement an efficient software barrier implementation [33], and incur a negligible overhead due to the synchronization instructions. The SSE code also **scales well**, and we achieve around 3.2X SP SSE scaling, and 1.65X DP SSE scaling.

GPU: Spatial blocking gives a large benefit of 2.8X over no-blocking for single precision data (Figure 4(c)). This is a result of the absence of caches on GPU. If data is not explicitly transferred from global to on-chip shared memory, then the same data is loaded multiple times for different neighbors. However, spatially blocked code is still bandwidth bound, and 3.5D blocking converts it to a compute bound kernel. This results in a performance gain of 1.9X-2X, resulting in an absolute performance of 17,100 million updates/second. For DP data, spatial blocking is in itself sufficient to convert it to a compute bound kernel since double precision GPU ops are far fewer than SP ops. Temporal blocking is then unnecessary for DP. The DP performance is 4,600 million updates/second.

B. LBM

LBM has very high bytes/op and is usually bandwidth bound even with spatial blocking.

CPU: On CPU (shown in Figure 4(a)), the no-blocking single precision LBM obtains a bandwidth of 20.5 GB/s, close to the maximum achievable. This number does not change with spatial blocking since LBM does not have spatial data-reuse - thus we do not consider this version. Performing only temporal blocking results in performance gains *only* for small data sets (for e.g. 64^3), where our buffer with $dim_X^{3.5D} = dim_Y^{3.5D} = 64$ fits in the LLC. For larger data sets, temporal blocking without spatial blocking does not help, since the necessary data in different XY slabs cannot be kept simultaneously in cache without spatial blocking.

Our 3.5D blocking results in compute-bound performance. The performance is 171 million updates/second for the 256^3 grid, with around 20% drop in performance due to the overestimation at the boundaries. The DP performance is about half the SP performance, since it uses twice as many flops as well as bandwidth.

GPU: No-blocking gets us a bandwidth bound performance of 485 million updates/second for SP. Since the GTX 285 does not have a large cache, we cannot maintain sufficient data across different XY slabs for effective temporal blocking. Hence temporal blocking cannot be done. LBM in DP is compute bound on the GPU even without blocking. We obtain about 39 DP Gops/second, which is within 15-20% of the peak compute bound number. Hence blocking will not improve performance.
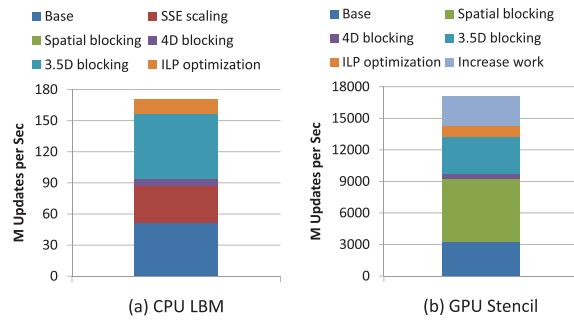
Fig. 5.   Breakdown of (a) LBM on CPUs (b) 7-point stencil on GPUs

C. Analysis of CPU and GPU performance

In this section, we provide a breakdown of the performance gains we achieved through our various optimizations. We focus on the GPU 7-point stencil and CPU LBM as illustrative examples. We use SP performance results in this section, and illustrate the cumulative effect of applying optimizations one after the other.

The breakdown for the CPU LBM stencil is given in Figure 5(a). The base performance is parallelized no-blocking scalar code (without SSE) which gives a performance of 52 million updates/second. This number is bound by compute, but does not use SSE units. However, using 4-wide SSE instructions (second bar) only improves performance to 87 million updates/second (and not by 4X) because the performance now becomes limited by memory bandwidth and not the number of instructions.

Performing spatial blocking alone (third bar) does not help at this stage, since there is no spatial data reuse and hence no reduction in memory bandwidth. Performing 4D blocking reduces the bandwidth due to temporal reuse of data. However, it introduces a large overhead of around 2.03X and the performance only improves by 8%. Performing 3.5D blocking reduces the overestimation, and we get a performance of 157 million updates/second. Finally, we perform optimizations to increase ILP (such as unrolling and software prefetching). This takes to performance to the final 171 million updates/second.

Figure 5(b) provides the breakdown for GPU 7-point stencil. We start with a performance of 3,300 million grid point updates/second for naive no-blocking code. As explained in Section VII-A, this results in unnecessary bandwidth use. The second bar is spatial blocking, which brings down the elements read to *about one per element* – there is a bandwidth overestimation of 13%. Spatial blocking yields a performance of 9,234 million updates/second. The third bar denotes 4D (3D spatial + 1D temporal) blocking, for comparison purpose with our 3.5D blocking scheme. 3D spatial blocking leads to a smaller blocking dimension, resulting in high overestimation, and the resultant 4D performance *only* improves to around 9,700 million updates/second (at improvement of 5%). The fourth bar is our 3.5D blocking scheme, which results in a performance of around 13,252 million updates/second. Once the kernel is made compute bound, we increase the

instruction level parallelism by loop unrolling. This gives us a performance of 14,345 million updates/second. The final step is then to decrease the number of instructions. We noticed that a number of instructions were spent in conditional checks for boundaries, computation of loop indices and so on, which are essentially per-thread overheads. We decreased the number of such instructions by making each thread perform more than one update, resulting in amortization of such overhead. This final step took our performance to 17,115 million updates/second.

D. Comparison with other stencil implementations

CPU: For 7-point stencil with DP, Datta [10] has reported the fastest performance of around 1,000 million updates on a single-socket 2.66GHz Intel Xeon X5550 with a bandwidth of around 16.5 GB/s. This performance number matches closely with our no-blocking bandwidth-bound number in Figure 4(b) after normalizing to our machine specs by scaling their number by the ratio of our and their memory bandwidth ($1000 * 22/16.5 = 1333$). In comparison, our 3.5D blocking performance is around 1,995 million updates/sec, an improvement of 1.5X over this normalized number. For 7-Point stencil with SP, the best reported performance is bandwidth bound, and cannot be improved unless using temporal blocking as in our 3.5D blocking scheme. Our performance using 3.5D blocking is around 4,000 million updates/second, an improvement of around 1.5X.

For LBM with DP, Habich et al. [13] report a performance of around 64 MLUPS on dual-socket (8-core) 2.66GHz Intel Nehalem. To normalize this number to our single-socket 3.2GHz Nehalem, we scale by 0.5 (for single socket) and then by 3.2GHz/2.66GHz (assuming compute bound) to get 38.5 MLUPS. In comparison, our 4-core number is around 80 MLUPS, which is around 2.08X faster than their implementation (using the normalized number). For LBM with SP, our 3.5D blocking improves the bandwidth bound performance of 87 million updates to around 180 million updates – an improvement of around 2.1X.

GPU: For 7-Point stencil with DP, Datta et al. [11] report a performance of around 4,500 million updates/sec on Nvidia GTX280. The reported performance is compute-bound, In comparison, our performance on GTX285 is around 4,600 million updates/sec, which is around 10-15% slower than their performance (normalizing to our platform). Note that we have not used any temporal blocking since the spatial blocking is close to compute bound and would not benefit from our 3.5D blocking scheme. For 7-Point stencil with SP, the performance is bandwidth bound without temporal blocking. Our 3.5D blocking scheme improves the performance by around 1.8X, and the resultant performance is within 30% of the peak compute-bound performance – the difference being because of overestimation of the compute due to the small block dimensions.

LBM with DP is compute bound with spatial blocking itself. Hence our 3.5D blocking does not improve it any

further. However, the SP variant is indeed bandwidth bound, but due to the small size of on-chip memory, the blocking dimensions are too small to give any speedup for this kernel.

VIII. Discussion

Future architecture trend of decreasing bandwidth-to-compute ratio (Γ): Intel Westmere CPU [34] has a lower Γ than the current Nehalem architecture, and this trend of decreasing peak bandwidth-to-compute ratio is expected to continue well into the future. Therefore our 3.5D blocking would become even more important for stencil based kernels – requiring larger temporal blocking to exploit the increasing compute resources. A larger temporal blocking also requires a proportionately larger on-chip cache to reduce the overhead of 3.5D blocking.

Small GPU Cache size: For stencil kernels with large $\mathcal{R}$ or $\mathcal{E}$, the size of shared memory and register file is too small for temporal blocking to provide speedups in runtime. For example, LBM SP is currently bandwidth bound, but requires an order of magnitude larger cache for 3.5D blocking to improve performance. Lastest and future GPUs (e.g. Fermi [9]) have a much larger cache than GTX285, and kernels like LBM SP should benefit from our blocking algorithm.

Double-Precision on GPU: Due to the low compute density of DP on GPU, most of the stencil kernels are compute-bound, and much slower than the corresponding SP performance (with 3.5D blocking). Since the GPU Fermi architecture is expected to increase the DP compute, we believe 3.5D blocking would be required for DP stencil kernels on GPU too. On CPU, since peak DP throughput is half of the peak SP throughput, we already see the benefit of 3.5D blocking for both the stencil kernels analyzed in this paper – LBM and 7-Point Stencil.

IX. Conclusions

In this paper, we propose and implement a 3.5D blocking scheme for stencil kernels by performing a 2.5D spatial and 1D temporal blocking to convert bandwidth bound kernels into compute bound kernels. The resultant stencil computation scales well with increasing core counts and is also able to exploit data-level parallelism using SIMD operations. Additionally, we provide a framework that determines the various blocking parameters – given the byte/op of the kernel, peak bytes/op of the architecture and the on-chip caches available to hold the blocked data. As a result, the performance of 7-point stencil and LBM is comparable or better than the fastest reported in the literature for both CPUs and GPUs. With the future architectural trend of increasing compute to bandwidth ratio, our 3.5D blocking scheme would become even more important for stencil based kernels.

References

[1] M. Berger and J. Oliger, "Adaptive mesh refinement for hyperbolic partial differential equations," *Journal of Computational Physics*, vol. 53, no. 1, pp. 484–512, 1984.

[2] R. Bleck, C. Rooth, D. Hu, and L. T. Smith, "Salinity-driven thermocline transients in a wind- and thermohaline-forces isopycnic coordinate model of the north atlantic," *Journal of Physical Oceanography*, vol. 22, no. 12, pp. 1486–1505, 1992.

[3] H. Dursun, K. ichi Nomura, L. Peng, R. Seymour, W. Wang, R. K. Kalia, A. Nakano, and P. Vashishta, "A multilevel parallelization framework for high-order stencil computations," in *Euro-Par*, 2009, pp. 642–653.

[4] A. Nakano, P. Vashishta, and R. K. Kalra, "Multiresolution molecular dynamics for realistic materials modeling on parallel computers," *Computer Physics Communications*, vol. 83, no. 1, pp. 197–214, 1994.

[5] L. Renganarayanan, M. Harthikote-Matha, R. Dewri, and S. V. Rajopadhye, "Towards optimal multi-level tiling for stencil computations," in *IPDPS*, 2007, pp. 1–10.

[6] F. Shimojo, R. K. Kalia, A. Nakano, and P. Vashishta, "Divide-and conquer density functional theory on hierarchical real-space grids: parallel implementationa and applications," *Physical Review*, vol. B, no. 77, pp. 1–12, 2008.

[7] "Intel Advanced Vector Extensions Programming Reference," http://softwarecommunity.intel.com/isn/downloads/intelavx/Intel−AVX−Programming−Reference−31943302.pdf, 2008.

[8] L. Seiler, D. Carmean, E. Sprangle, T. Forsyth, M. Abrash, P. Dubey, S. Junkins, A. Lake, J. Sugerman, R. Cavin, R. Espasa, E. Grochowski, T. Juan, and P. Hanrahan, "Larrabee: a many-core x86 architecture for visual computing," *ACM Trans. Graph.*, vol. 27, no. 3, pp. 1–15, August 2008.

[9] Nikolaj Leischner and Vitaly Osipov and Peter Sanders, "Fermi Architecture White Paper," 2009. [Online]. Available: http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf

[10] K. Datta, "Auto-tuning stencil codes for cache-based multicore platforms," Ph.D. dissertation, EECS Department, University of California, Berkeley, Dec 2009. [Online]. Available: http://www.eecs.berkeley.edu/Pubs/TechRpts/2009/EECS-2009-177.html

[11] K. Datta, M. Murphy, V. Volkov, S. Williams, J. Carter, L. Oliker, D. Patterson, J. Shalf, and K. Yelick, "Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures," in *SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*. Piscataway, NJ, USA: IEEE Press, 2008, pp. 1–12.

[12] M. Frigo and V. Strumpen, "The memory behavior of cache oblivious stencil computations," *J. Supercomput.*, vol. 39, no. 2, pp. 93–112, 2007.

[13] J. Habich, T. Zeiser, G. Hager, and G. Wellein, "Enabling temporal blocking a lattice boltzmann flow solver through multicore-aware wavefront parallelization," *21st International Conference on Parallel Computational Fluid Dynamics*, pp. 178–182, 2009.

[14] S. Kamil, K. Datta, S. Williams, L. Oliker, J. Shalf, and K. Yelick, "Implicit and explicit optimizations for stencil computations," in *MSPC '06: Proceedings of the 2006 workshop on Memory system performance and correctness*. New York, NY, USA: ACM, 2006, pp. 51–60.

[15] P. Micikevicius, "3d finite difference computation on gpus using cuda," in *GPGPU-2: Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units*. New York, NY, USA: ACM, 2009, pp. 79–84.

[16] G. Rivera and C.-W. Tseng, "Tiling optimizations for 3d scientific computations," in *Supercomputing '00: Proceedings of the 2000 ACM/IEEE conference on Supercomputing (CDROM)*. Washington, DC, USA: IEEE Computer Society, 2000, p. 32.

[17] G. Wellein, G. Hager, T. Zeiser, M. Wittmann, and H. Fehske, "Efficient temporal blocking for stencil computations by multicore-aware wavefront parallelization," in *COMPSAC '09: Proceedings of the 2009 33rd Annual IEEE International Computer Software and Applications Conference*. Washington, DC, USA: IEEE Computer Society, 2009, pp. 579–586.

[18] S. Williams, J. Carter, L. Oliker, J. Shalf, and K. Yelick, "Optimization of a lattice boltzmann computation on state-of-the-art multicore platforms," *J. Parallel Distrib. Comput.*, vol. 69, no. 9, pp. 762–777, 2009.

[19] K. Datta, S. Kamil, S. Williams, L. Oliker, J. Shalf, and K. Yelick, "Optimization and performance modeling of stencil computations on modern microprocessors," *SIAM Rev.*, vol. 51, no. 1, pp. 129–159, 2009.

[20] T. Pohl, M. Kowarschik, J. Wilke, K. Iglberger, and U. Rde, "Optimization and profiling of the cache performance of parallel lattice boltzmann codes in 2d and 3d," *PARALLEL PROCESSING LETTERS*, vol. 13, no. 4, pp. 549–560, 2003.

[21] S. Williams, J. Shalf, L. Oliker, S. Kamil, P. Husb, and K. Yelick, "Scientific computing kernels on the cell processor," *International Journal of Parallel Programming*, vol. 35, p. 2007, 2007.

[22] M. Wittmann, G. Hager, and G. Wellein, "Multicore-aware parallel temporal blocking of stencil codes for shared and distributed memory," *LSPP10: Workshop on Large-Scale Parallel Processing at IPDPS*, 2010.

[23] J. Treibig, G. Wellein, and G. Hager, "Efficient multicore-aware parallelization strategies for iterative stencil computations," *Submitted to Computing Research Repository (CoRR)*, vol. abs/1004.1741, 2010.

[24] P. Bailey, J. Myre, S. Walsh, D. Lilja, and M. Saar, "Accelerating lattice boltzmann fluid flow simulations using graphics processors," in *ICPP-2009: 38th International Conference on Parallel Processing*, Vienna, Austria, 2009.

[25] A. Kaufman, Z. Fan, and K. Petkov, "Implementing the lattice boltzmann model on commodity graphics hardware," *Journal of Statistical Mechanics: Theory and Experiment*, vol. 2009, June 2009.

[26] F. Kuznik, C. Obrecht, G. Rusaouen, and J.-J. Roux, "Lbm based flow simulation using gpu computing processor," *Computers & Mathematics with Applications*, vol. 59, no. 7, pp. 2380 – 2392, 2010, mesoscopic Methods in Engineering and Science, International Conferences on Mesoscopic Methods in Engineering and Science. [Online]. Available: http://www.sciencedirect.com/science/article/B6TYJ-4X9D5D0-3/2/9e7676667251dd6bdc7ea63fbc0232a8

[27] L. Peng, K.-I. Nomura, T. Oyakawa, R. K. Kalia, A. Nakano, and P. Vashishta, "Parallel lattice boltzmann flow simulation on emerging multi-core platforms," in *Euro-Par '08: Proceedings of the 14th international Euro-Par conference on Parallel Processing*. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 763–777.

[28] E. Riegel, T. Indinger, and N. A. Adams, "Implementation of a lattice-boltzmann method for numerical fluid mechanics using the nvidia cuda technology," *Computer Science – Research and Development*, vol. 23, no. 3-4, pp. 241–247, 2009.

[29] J. Tolke, "Implementation of a lattice boltzmann kernel using the compute unified device architecture developed by nvidia," *Comput. Vis. Sci.*, vol. 13, no. 1, pp. 29–39, 2009.

[30] M. Reilly, "When multicore isn't enough: Trends and the future for multi-multicore systems," in *HPEC*, 2008.

[31] "Intel SSE4 programming reference," 2007, http://www.intel.com/design/processor/manuals/253667.pdf.

[32] NVIDIA, "NVIDIA CUDA TM Programming Guide, Version 3.0," 2010. [Online]. Available: http://download.intel.com/pressroom/kits/32nm/westmere/Intel_32nm_Overview.pdf

[33] J. M. Mellor-Crummey and M. L. Scott, "Algorithms for scalable synchronization on shared-memory multiprocessors," *ACM Trans. Comput. Syst.*, vol. 9, no. 1, pp. 21–65, 1991.

[34] Intel Corporation, "Introduction to Intel's 32nm Process Technology," 2009.