Efficient temporal blocking for stencil computations by multicore-aware wavefront parallelization

Gerhard Wellein, Georg Hager, Thomas Zeiser, Markus Wittmann
Erlangen Regional Computing Center (RRZE)
University of Erlangen-Nuremberg
Erlangen, Germany
Email: gerhard.wellein@rrze.uni-erlangen.de

Holger Fehske
Institute for Theoretical Physics
University of Greifswald
Greifswald, Germany
Email: fehske@physik.uni-greifswald.de

***Abstract*—We present a pipelined wavefront parallelization approach for stencil-based computations. Within a fixed spatial domain successive wavefronts are executed by threads scheduled to a multicore processor chip with a shared outer level cache. By re-using data from cache in the successive wavefronts this multicore-aware parallelization strategy employs temporal blocking in a simple and efficient way. We use the Jacobi algorithm in three dimensions as a prototype for stencil-based computations and prove the efficiency of our approach on the latest generations of Intel's x86 quad- and hexa-core processors.**

***Keywords*-temporal blocking; wavefront parallelization; stencil computations; multicore;**

I. Introduction

Stencil computations can be found at the core of many scientific and technical applications based on regular lattices. For the important class of partial differential equation (PDE) solvers they are a key performance factor. This does not only hold for serial applications but is also true for massively parallel large scale multigrid PDE solvers (see e.g. [1]), where the time-consuming smoothing steps are frequently composed of stencil computations such as red-black Gauss-Seidel or Jacobi schemes. As these involve only nearest neighbors for a stencil update, standard optimization techniques like spatial blocking are usually applied to ensure optimal spatial locality of data accesses.

It has been shown recently [2] that for state-of-the-art multicore architectures a near to optimal stencil implementation requires elaborate optimization and auto-tuning, even if more complex temporal blocking techniques are ignored. Conventional temporal blocking performs multiple updates on a small block of the computational domain before proceeding to the next block. The size of the blocks have to be carefully chosen to fit into the processor's cache. Apart from having a machine dependent tuning parameter, this kind of temporal blocking in three spatial dimensions has been found to not deliver performance improvements [3] because it generates rather short loops resulting in substantial performance penalties from pipeline start-up effects and, even worse, from strongly restricted data prefetching abilities. Cache oblivious algorithms as proposed by Frigo et al. [4] are hardware independent but come at the cost of irregular block access patterns, which cause many data TLB misses. This was shown for a 3D lattice Boltzmann (LB) application kernel in Ref. [5].

In this report we propose a wavefront-based thread level parallelization scheme which enables temporal blocking for stencil computations in multicore environments with shared caches. The basic idea is to run multiple wavefronts through the computational domain at the same time but appropriately shifted in space (depending on the stencil used). Each wavefront represents an update step of the lattice and is executed by a single thread. Binding all threads that run successive wavefronts for the same computational domain to a single multicore chip with a shared cache restricts data access to main memory to a load operation for the initial wavefront and a store operation for the final wavefront; all other (intermediate) data accesses can be satisfied from the shared cache. To make efficient use of multi-chip environments we also implement a simple spatial domain decomposition which assigns appropriately sized blocks to the available multicore chips. The proposed scheme is tailored to multicore architectures with shared caches — which will be the major design principle for most standard architectures in the years to come — and does not exhibit the drawback of very short loop lengths. Wavefront parallelization schemes are well known from massively parallel applications [6]. However, the use of the wavefront approach on the thread level to enable temporal blocking by exploiting the unique feature of shared caches in multicore processors is new to our knowledge.

In order to compare our results with a recent study by Datta et al. [2] we have chosen the 3D Jacobi method, which serves as a paradigm for stencil computations and can, e.g., be used to solve the explicit heat equation. Temporal blocking is known to be most beneficial if the single core performance can already saturate most of the available main memory bandwidth, i.e. if there is only very limited parallel scalability for memory-bound loops within a multicore chip. Thus we focus on Intel-based multi-socket platforms using

0730-3157/09 $25.00 © 2009 IEEE
DOI 10.1109/COMPSAC.2009.82

		Clovertown	**Nehalem**	**Dunnington**
Type		Xeon 5345 @2.33 GHz	"Core i7" @2.66 GHz	Xeon 7460 @2.66 GHz
L1 group	size [kB]	32	32	32
	TRIAD GB/s	3.9	11.6	3.0
L2 group	L2 size [MB]	4	0.25	3
	# cores	2	1	2
	TRIAD GB/s	4.0	see L1	3.5
L3 group / socket	L3 size [MB]	-	8	16
	# cores	4	4	6
	TRIAD GB/s	4.0	16.6	3.5
System	# sockets	2	2	4
	raw bw [GB/s]	21.3	51.2	34.0
	TRIAD GB/s	7.8	32.7	13.2

Table I
OVERVIEW ON CACHE GROUP STRUCTURE AND STREAM TRIAD (ARRAY SIZE OF 20,000,000 ELEMENTS) PERFORMANCE FOR THE SYSTEMS IN THE TEST-BED. NON-TEMPORAL STORES WERE USED THROUGHOUT, SO ALL BANDWIDTH NUMBERS DENOTE ACTUAL BUS TRAFFIC.

state of the art quad-core (”Clovertown” / ”Nehalem”) and hexa-core (”Dunnington”) variants.

II. EXPERIMENTAL TEST-BED

The key architectural features of the compute nodes evaluated in this report are presented in Tab. I. Since stencil computations are data intensive, the attainable main memory bandwidth as measured with optimized stream benchmark [7] runs are shown as well. “Optimized” refers to the use of non-temporal stores, which are important to get unbiased results on x86 architectures (cf. the discussion in the next section). Note that we focus on the cache levels available for data only, ignoring all instruction caches.

Regarding the complex organization of modern multi-core processors we introduce a convenient terminology to express which cache level is shared by how many cores: We call a group of cores that shares a certain cache a *cache group.* Thus there are L1 groups (which consist of a single core on all current multi-core designs), L2 groups, etc.. There can be multiple instances of any cache group on a multi-core chip, and it is also common to have outermost cache groups comprising less than the number of cores in a socket (see Sect. II-A below for an example).

For all tests the Intel compilers in version 11.0.069 were used.

A. Intel Xeon E5345 (Clovertown)

Although the Clovertown CPU is considered to be Intel's first quad-core processor it is mainly based on proven dual-core technology: Two dual-core Xeon chips reside on a multi-chip module (MCM), sharing no resources other than the front side bus (FSB). The chips are clocked at 2.33 GHz providing a double precision peak performance of 9.32 GFlops per core. Each core has its dedicated 32 KB L1 cache, and the two 4 MB L2 groups comprise two cores each.

Within the compute node each of the two sockets carries a quad-core MCM and delivers a raw FSB bandwidth of 10.66 GB/s (FSB1333). The chipset connects the two FSBs to four fully buffered DDR2-667 DRAM channels, exactly matching the aggregated FSB bandwidth (21.33 GB/s). Unfortunately the bus architecture of the Intel-based systems is known to be very inefficient in terms of sustainable bandwidth, which can be seen from the STREAM numbers in Table I, where less than 50 % of the theoretical bandwidth can be achieved in the STREAM TRIAD test case.

B. Intel Xeon X7460 (Dunnington)

This node is still based on an FSB architecture. A single FSB1066 (8.5 GB/s) connects each socket with the chipset, matching eight channels of fully buffered DDR2-533 interfaces to the memory DIMMs for a total of 34 GB/s of theoretical bandwidth. A socket contains a hexa-core chip running at 2.66 GHz, which provides a single socket double precision floating point performance of 63.84 GFlops (6 cores $\times 4$ Flops/(core·cycle) $\times 2.66$ Gcycle/s) resulting in a single node performance 0.25 TFlops.

The most interesting detail with respect to this work is the hierarchical hexa-core design. Similar to the Clovertown, the basic building block of the chip is again a dual-core, 3 MB L2 group with dedicated L1 for each core. In contrast to the Clovertown, however, there is a large 16 MB L3 group containing all six cores. Advanced multicore-aware programming techniques should address this outer hierarchy level to reduce data accesses to main memory, since the system balance of 0.136 Bytes/Flop (34 GB/s/250 GFlops) is expected to be a typical number in future many-core architectures.

C. Intel Nehalem

The Nehalem processor architecture ("Core i7") will be Intel's x86 flagship in the upcoming years. Apart from implementing a native quad-core, the important step forward is the complete redesign of the memory subsystem. Multi-socket servers feature a cache-coherent non-uniform memory access (ccNUMA) architecture, alleviating the limited scalability and efficiency of the FSB. Similar to HyperTransport as used in multi-socket AMD compute nodes, "QuickPath" Interconnect (QPI) channels provide fast paths for inter-socket communication and cache coherence traffic without interfering with local memory access. Moreover, the single socket memory bandwidth has been substantially improved as well by introducing an on-chip memory controller which drives three DDR3 memory channels and allows for a per-socket bandwidth of approximately 32 GB/s if DDR3-1333 DIMMs are used.

The dual-socket "Nehalem-EP" node used in this report is a preproduction system. We expect production hardware to deliver similar performance levels. The quad-core processors run at 2.66 GHz, delivering a single core double precision performance of 10.64 GFlops. The 32 kB L1 and 256 kB L2 groups comprise a single core each, whereas the 8 MB L3 group encompasses all four cores. The node is equipped with twelve 1 GB DDR3-1066 DIMMs. Note that the STREAM numbers of a single node (see Table I) are similar to those of a single NEC SX6 vector processor as used in the Earth Simulator.

The Nehalem is also capable of running two threads per core very efficiently through the Simultaneous Multi-Threading (SMT) feature. It has been much improved compared to Intel's previous Hyper-Threading technique and first reports have already demonstrated significant application performance improvements through SMT [8]. In this report we do not use SMT.

III. Baseline Implementation and Optimizations

The Jacobi algorithm can be implemented in 3D as a triple loop nest traversing the complete computational domain by updating each grid point (we use Fortran notation):

```
do k = 1 , Nk
 do j = 1 , Nj
  do i = 1 , Ni
   y(i,j,k) = a * x(i,j,k) + b * (
                x(i-1,j,k) + x(i+1,j,k) +
                x(i,j-1,k) + x(i,j+1,k) +
                x(i,j,k-1) + x(i,j,k+1) )
  enddo
 enddo
enddo
```

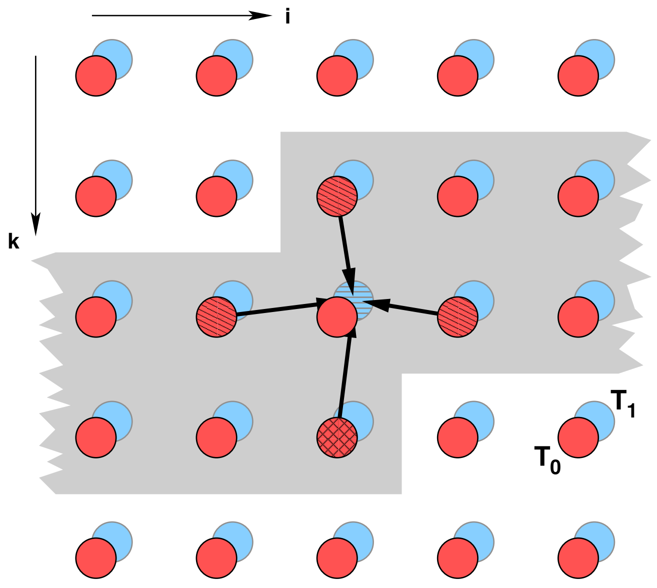

Figure 1. Schematic view of the Jacobi iteration in 2D using a five-point stencil. The shaded region marks the two lines (planes in 3D) of grid points that need to stay in cache to fully reuse each entry of x from cache.

This is a single Jacobi "sweep" that updates the computational domain once, using a seven-point stencil. For iterative solvers or smoothing algorithms, an outer loop performs this iteration several times. Seemingly, one lattice site update executes eight floating point operations (Flops) and transfers 16 Bytes (two double precision words) between main memory and cache, assuming the cache is large enough to hold at least two successive $N_j \times N_i$ planes for array x. Figure 1 illustrates this requirement schematically for the 2D case. A peculiarity of cache-based architectures leads to a significant increase in data transfer for memory-bound situations: Usually all communication between cache and memory is handled in units of cache lines. A store miss to a cache line for y will hence cause a "Read for Ownership" (RFO) before the line can be modified in cache and later evicted to memory. Consequently, each store to y moves 16 Bytes instead of 8 Bytes, which leads to 24 Bytes of overall data transfer. X86 architectures like the ones considered here provide a way to avoid the RFO and write directly to memory via a write combine buffer; the special instructions required for this are called *non-temporal stores*.

In the following we assume a cubic domain ($N_k = N_j = N_i = N$) and focus on memory-bound problems, where the overall data set is much larger than any cache size. A convenient performance measure for stencil-based computations is the lattice update rate given in million lattice site updates per second (MLUPs). For a single update of the whole lattice in the pseudo-code presented above the performance is given by $N^3/(10^6 \cdot \text{walltime [sec]})$. The time for a single lattice

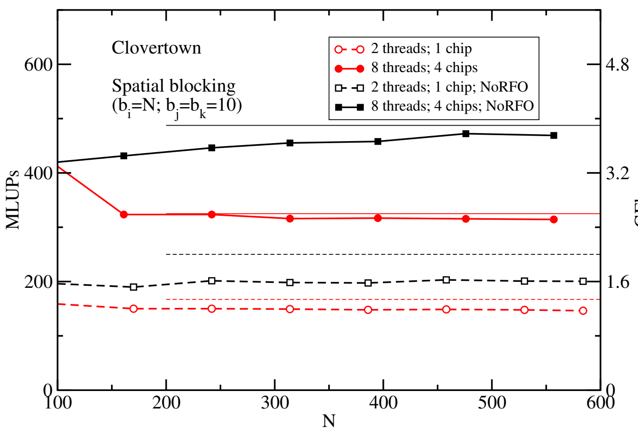

Figure 2. Clovertown system: Single chip and single node performance including spatial blocking with and without the RFO (NoRFO). The thin horizontal lines starting at $N = 200$ show the upper performance limits as determined from the stream bandwidth.

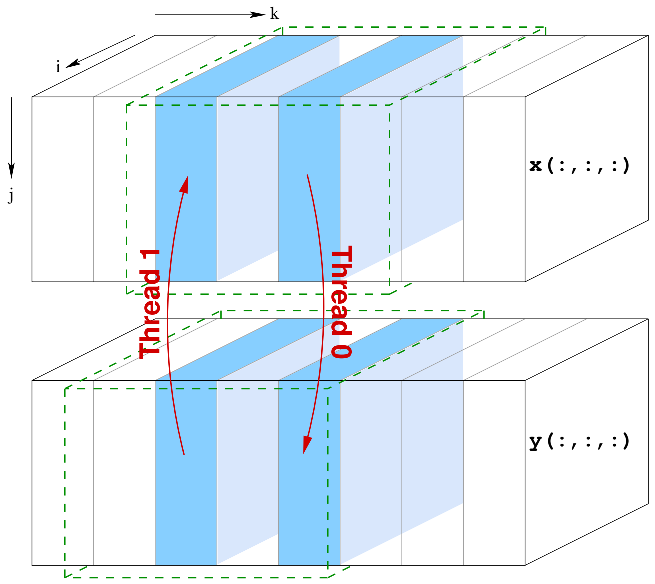

Figure 3. Time blocking through pipeline parallel processing by a two-thread wavefront group. Dashed boxes indicate (i,j) layers that must be kept in cache for optimal reuse of cache lines.

site update is

$$T_0 = \frac{16\,\text{Byte}}{B_M} \ , \qquad (1)$$

where B_M is the attainable STREAM bandwidth as given in Table I assuming the RFO has been suppressed through the use of non-temporal stores. For the Clovertown system this yields an upper limit of 250 MLUPs for one L2 group, i.e. two cores on a single chip, and of 487.5 MLUPs for the complete compute node, i.e. eight cores. If non-temporal stores are not employed those numbers are reduced to 167 MLUPs and 325 MLUPs, respectively. Figure 2 presents performance data for a standard implementation of the code snippet above plus spatial blocking with appropriate blocking factors (b_i, b_j, b_k) as suggested in Ref. [2]. Shared memory parallelization of the outer k-loop was implemented with a standard OpenMP loop worksharing directive, and data locality for ccNUMA architectures was ensured by appropriately parallelizing the initialization loops for arrays `x` and `y` ("first touch"). Including the RFO, the single chip/node performance is close to the maximum as estimated above and is in line with the data presented in Ref. [2] (2.3 GFlops full node performance) for a similar hardware configuration. Using appropriate compiler intrinsics, the RFO can be suppressed (NoRFO), which leads to a corresponding performance improvement. The resulting performance level is substantially higher than the auto-tuning results of Ref. [2], where cache bypass only yielded a 10 % (2.5 GFlops) boost.

IV. Temporal Blocking by Wavefront Parallelization

In view of bandwidth-starved but cache-abundant multicore designs like Intel's Clovertown and Dunnington chips, the need arises to reformulate stencil algorithms in order to lessen the pressure on the memory subsystem. Although a single Jacobi sweep over the full lattice is limited by memory bandwidth, real-world implementations often perform multiple iterations. The basic idea for performance optimization is hence to re-use data (e.g., one complete lattice layer) that has just been moved to the cache by one core and perform another iteration on it "as soon as possible", using another core in the same cache group. During this second update, the first core can already perform the first time step on the next layer. Three conditions must hold for this scheme to work properly:

1) The second core may only modify data that is sufficiently far away from data currently needed by the first core in order to avoid race conditions. The minimum distance is of course determined by the choice of the stencil.
2) The lattice site data to be used for the second iteration must still reside in the shared cache.
3) Both threads must be pinned to cores in the same cache group. We use a shared object to overload the `pthread_create()` library call and employ `sched_setaffinity()` to pin each thread in turn, skipping OpenMP "shepherd" threads [9].

Figure 3 shows schematically how temporal blocking by *pipelined wavefronts* works: Thread 0, running on one core in the cache group, updates layer `y(:,:,k)`, requiring input data from `x(:,:,k-1:k+1)`. Note that at this time, `x(:,:,k-1:k)` has already been moved to cache in the previous two `k` iterations. Since the 3D seven-point stencil only contains next-neighbor dependencies, a single site update involves three adjacent `k` layers. As can be seen in Fig. 3, a concurrent second update can be performed by thread 1 on `x(:,:,k-2)` based on input data from `y(:,:,k-3:k-1)`. Of course after each k iteration all threads must be synchronized to prevent race conditions.

If the cache is large enough to hold four successive (i,j) layers of each `x` and `y` (dashed boxes in Fig. 3), the overall data transfer requirement to and from memory can be reduced by one third: instead of two loads and one store for a single site update (as required by a naive, spatially blocked implementation including RFO), we now need two loads (load on `x` and RFO on `y`) and two stores (cache line evictions on `x` and `y`) for two updates. However, there is even more room for improvement. As two full updates are now performed concurrently in one step, and the result is stored back into `x`, most of array `y` is redundant as it serves only as a temporary buffer for the results of the first update. Therefore `y` can be replaced by a temporary array `tmp(:,:,0:3)`, represented by the lower dashed box in Fig. 3. The following pseudo-code describes the concurrent update of planes k and $k-2$ by thread 0 and thread 1, respectively:

```
thread 0:
tmp(:,:,mod(k,4))<-- x(:,:,k-1:k+1)

thread 1:
x(:,:,k-2)<-- tmp(:,:,mod(k-3,4):mod(k-1,4))
```

Apart from a reduction of nearly a factor of two in memory requirements, this optimization saves all main memory transfers to and from array `y`, leaving one load and one store (eviction) for two site updates. Compared to the original algorithm, this is an improvement in computational intensity by a factor of three (two if RFO can be avoided on the standard code) leading to corresponding performance gains. This assumes that (i) the shared cache is large enough to hold both dashed boxes and (ii) the caches are infinitely fast. The impact of finite cache bandwidth can be estimated using a simple model. If the wavefront approach introduces x additional on-chip eight-byte transfers (between registers and outer level caches) that can not be overlapped with memory traffic, the overall transfer time for two site updates is:

$$T = T_0 + x \cdot \frac{8\,\text{Byte}}{B_C} = T_0\left(1 + \frac{x}{2}\frac{B_M}{B_C}\right) \qquad (2)$$

Here T_0 is the same as in (1) and B_C denotes the outer level cache bandwidth as obtained by suitable STREAM-like bandwidth measurements. For the two quad-core processors considered in this report the ratio B_C/B_M is approximately 12 (Clovertown) and 4 (Nehalem) [10]. The maximum speed-up of the wavefront approach as compared to the standard NoRFO version is then $2T_0/T$. Owing to the complexity of modern multi-level (shared) cache architectures a quantitative determination of x is difficult. In the most favorable case, each k-layer updated by thread 0 needs to be transfered to the other thread via the shared outer-level cache, leading to $x = 2$. In reality x can be larger depending, e.g. on additional coherence overhead.

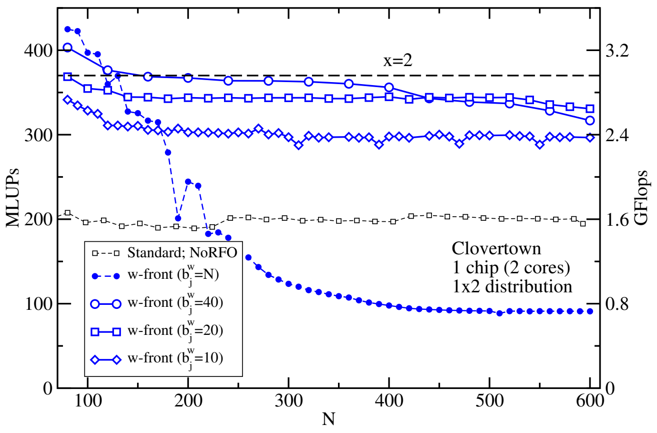

Figure 4. Clovertown system: Single chip (2 cores) performance running two wavefronts and using different blocking factors in j-direction. For comparison, the results of the standard version with non-temporal stores (from Fig. 2) are shown (NoRFO). The dashed line ($x = 2$) denotes the upper performance bound as given by (2) and based on a 200 MLUPs baseline of the standard version.

A straightforward generalization of the algorithm uses t_b instead of two threads so that one complete sweep performs t_b updates on every site. We then speak of a t_b-way wavefront group. Accordingly, $t_b - 1$ temporary arrays are needed to store intermediate steps, resulting in an overall cache requirement of $4t_b$ layers of size $N_i \times N_j$. A reliable performance model for the case $t_b > 2$ would require a much more detailed analysis of cache architectures and is beyond the scope of this work.

A. Single chip results

Figure 4 shows performance data for running two pipelined wavefronts ($t_b = 2$) on one L2 group of the Clovertown system (filled circles) in comparison to the standard NoRFO version which was shown in Fig. 2. Although the pipelined version outperforms the standard code, as expected, by a factor of two for problem sizes smaller than 100^3, performance breaks down quickly as N gets larger. This occurs because the cache size is not sufficient to accommodate eight (i,j) layers, and the second thread cannot work exclusively from cache. For achieving high performance on arbitrarily large problems, the complete domain should be split into sub-blocks of size $N_i \times b_j^w \times N_k$ along the j coordinate, reducing the cache requirements by a factor of N_j/b_j^w. Each sub-block is then completely updated by both wavefronts before they turn to the next sub-block in positive j direction. Blocking in i direction is an option, but small inner loops tend to cause inefficient utilization of the memory interface since they interfere with hardware prefetching [2]. Therefore we employed j blocking only.

Of course, the interfaces between sub-blocks require some sophisticated handling of boundary conditions. To update a

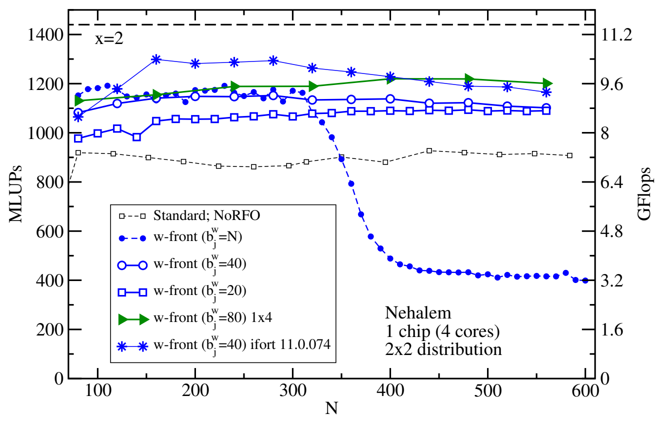

Figure 5. Nehalem system: Single chip (4 cores) performance running two wavefronts (2x2 distribution) and using different blocking factors in j-direction. The results for running four wavefronts using a blocking factor of $b_j^w = 80$ is shown as well (1x4).The dashed line ($x = 2$) denotes the upper performance bound as given by (2) and based on a 900 MLUPs baseline of the standard version.

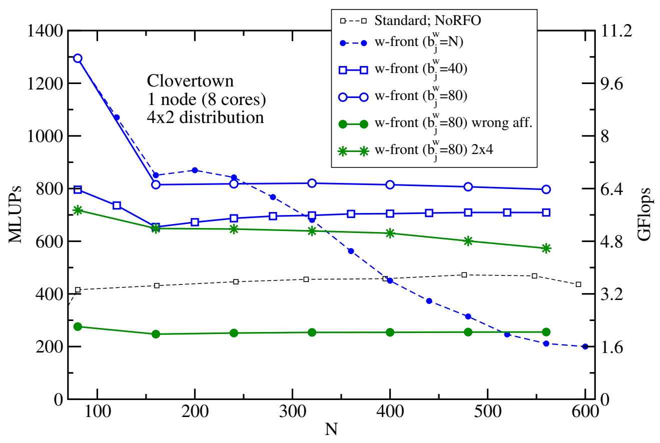

Figure 6. Clovertown system: Single node (8 cores) performance for the wavefront implementation with appropriate blocking factors in j-direction. The impact of inadequate choice of wavefront decomposition (2x4) and thread binding (wrong binding) is demonstrated. For comparison, the results of the standard version with non-temporal stores (from Fig. 2) are shown (NoRFO).

sub-block correctly, two additional arrays of size $t_b \times N_k \times N_i$ are used to hold initial data coming from the previous sub-block and to store initial data for the next. For a two-thread wavefront group as considered so far, one array would be sufficient. However we also want to support multiple concurrent wavefront groups traversing the same sub-block, where using a single boundary array would incur race conditions.

Figure 4 demonstrates the benefit of j-blocking for $b_j^w \in \{10, 20, 40, N\}$ (where $b_j^w = N$ corresponds to no blocking at all). Obviously, blocking can shift the performance breakdown due to cache size to larger grid dimensions. Two competing effects influence the optimal choice of b_j^w: A small block size reduces the cache requirement but increases boundary handling overhead. Thus b_j^w should be chosen just large enough to fit eight ($4t_b$) $N_i \times b_j^w$ layers comfortably into the shared cache. As can be seen from Fig. 4, $b_j^w = 40$ is a reasonable choice up to $N \lesssim 600$ where the cache requirement is roughly 1.6 MB (just below half of the shared cache size). Note that the overall memory consumption for a 600^3 lattice is already about 1.7 GB. For even larger problems, b_j^w should be reduced accordingly. Over the complete range of domain sizes shown in Fig. 4, the wavefront algorithm outperforms the best standard version by a factor between 1.5 and 1.8, which is in very good agreement with the simple performance model derived in Section IV.

Considering the Nehalem system one should be aware that one chip comprises four cores, the outer level (L3) cache is twice as large, and the attainable memory bandwidth is a factor of 3–4 larger compared to Clovertown. With four cores, a 4-way wavefront ($t_b = 4$) is the natural way to employ wavefront parallelization. Figure 5 shows performance data for $t_b = 4$ and $b_j^w = 80$ (filled triangles), which is a reasonable choice with respect to the cache size. As predicted by our performance model, the overall benefit compared to the standard algorithm ($\lesssim 1.3$) is smaller than for Clovertown because of the substantially lower B_C/B_M ratio of Nehalem. However, the discrepancy between model and measurements with the standard compiler was significantly larger than on Clovertown, suggesting potentials for further in-cache code optimizations. This conjecture is substantiated by a notable performance improvement when using the latest Intel compiler (version 11.0.074).

There is, however, an alternative choice for parallelization where a sub-block is traversed synchronously by two groups of two-way wavefronts (2×2). In this case a sub-block is decomposed further into two parts along the j direction, each getting two concurrent updates ($t_b = 2$). One would expect this approach to show significantly worse performance than the $t_b = 4$ version because fewer updates are carried out in cache. However, as Fig. 5 shows, the difference is rather marginal for a suitable choice of b_j^w. This can be attributed to the fact that a single Nehalem core can only utilize about 60 % of a socket's memory bandwidth (see Table I), leaving substantial headroom for a second memory-bound thread.

B. Full node results

It should be clear that affinity mechanisms must be employed in order to prevent threads from leaving their initial cache groups, thereby destroying temporal locality. When running wavefront parallelization on multiple cores that are not part of a cache group, special care must be taken of appropriate thread binding and decomposition into wavefront groups. Filled circles in Fig. 6 show results on the Clovertown system with four two-wavefront groups ($t_b = 2$), where each group was spread across cores in different L2 groups. As there is no L3 group, the temporary array must

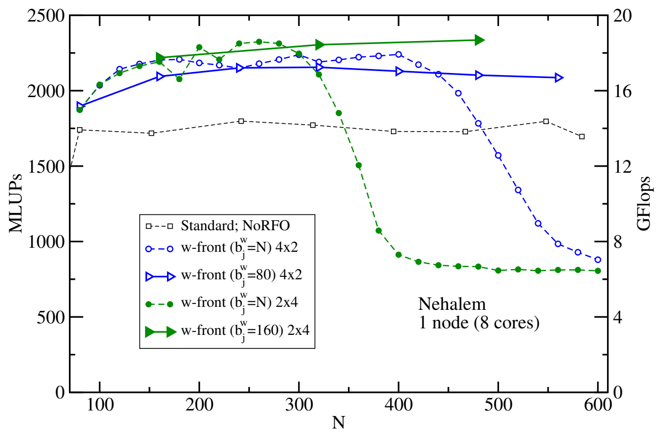

Figure 7. Nehalem system: Single node (8 cores) performance for the wavefront implementation with appropriate blocking factors in j-direction.

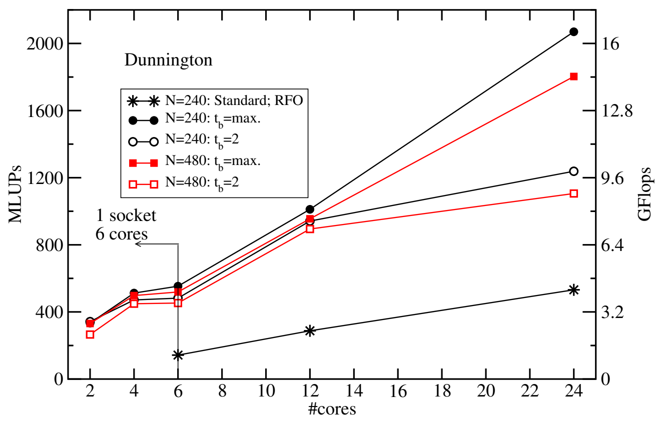

Figure 8. Dunnington system: Intranode scalability of the wavefront approach for two different domain sizes. Appropriate blocksizes have been chosen from the subset $40, 80, 120, 240$. One block has been assigned to a single socket (L2 group) if the number of wavefronts is maximum (2). For comparison, the results of the standard version including the RFO are shown.

reside in main memory and all updates are memory-bound. On the other hand, choosing $t_b = 4$ with two wavefront groups (stars) is not optimal as well because there is no cache group that comprises four cores. Thus after two updates to the current block, data has to be moved between the two L2 groups. This can only occur via cache line invalidations and a detour through main memory, effectively leading to a 4×2 setup with some additional overhead. Note that we have bound the first thread pair of each wavefront group to an L2 group, and the second pair to the other L2 group of the same socket. Choosing a round robin distribution would completely eliminate the shared cache benefit and result in a further substantial performance drop.

Due to the large (16 MB) aggregated cache on the Clovertown system, the typical performance breakdowns for large N can only be prominently seen at $b_j^w = N$, i.e. without j-blocking (filled circles with dashed line). With $b_j^w = 80$ (open circles) or 40 (open squares), respectively, the same behavior as in the one-chip results can be identified: A small block size leads to larger overhead for small N, but shifts the inevitable breakdown to larger N. Best overall performance is roughly doubled versus the single chip (see Fig. 4), reflecting the STREAM bandwidth scaling as shown in Table I.

Figure 7 shows full-system Nehalem results. A comparison with the single-chip data in Fig. 5 shows that main memory bandwidth scales across sockets in this ccNUMA system, since first touch placement was employed. Without j-blocking, the cache requirement of the wavefront algorithm is proportional to N^2, so we expect the 4-way wavefront performance to break down at a value of N that is a factor of $\sqrt{2}$ smaller than with the 2×4 setting. This is indeed confirmed by our measurements (filled and open circles for 2×4 and 4×2 wavefronts, respectively). With optimal block sizes for the range of N considered here, overall performance again shows similar characteristics as for the single chip: Although one would expect the 2×4 algorithm to be considerably slower than 4×2 due to the smaller number of updates performed in cache, they are not much different in practice because of the inability of the "front thread" to utilize the full memory bandwidth on a socket.

As an example for a heavily "bandwidth-starved" design, which may be prototypical for future multicore chips, we finally present strong scaling results for the four-socket hexa-core Dunnington system. Note that this is not a typical node to be found in commodity HPC clusters today. Two of the six cores on a socket can fully utilize the local memory bandwidth (see the STREAM data in Table I), leaving a lot of potential for temporal blocking techniques. Figure 8 shows performance data for two problem sizes ($N = 240$ and $N = 480$, respectively; circles vs. squares) and comparing two-way (open symbols) versus six-way (filled symbols) wavefronts. In all cases, appropriate block sizes have been chosen for best performance.

For small numbers of cores (up to two sockets), the benefit of using six-way wavefronts is rather small because (i) two threads can still draw significantly more bandwidth out of a socket than one and (ii) L3 cache bandwidth becomes a bottleneck if five out of six threads work exclusively on data in this cache level. However, as can be seen from the STREAM numbers in Table I, system bandwidth does not scale to four sockets. Hence at 24 threads, two wavefronts are strongly bandwidth-limited whereas six wavefronts make much better use of the computational resources (filled vs. open symbols). The seemingly superlinear scaling for $N = 240$ (filled circles) when going from 12 to 24 cores is an artifact of using different j blocking factors. Indeed no spatial blocking at all was employed for the full node run, avoiding the associated boundary exchange overhead.

The performance impact of the latter is visible from the difference to the 24 thread result at $N = 480$ where two blocks ($b_j^w = 240$) were used. For comparison, we have included data for the standard spatially blocked algorithm (stars), this time including RFO. For reasons yet unknown, suppressing the RFO with non-temporal stores on more than two sockets leads to exceptionally bad performance on Dunnington. The maximum performance of an efficient NoRFO version would be less than 750 MLUPs for the full node.

V. Conclusion

We have shown that pipelined wavefront parallelization is a way to efficiently implement temporal blocking for stencil-based algorithms on Intel's current multicore processors. As opposed to other temporal blocking schemes, our algorithm makes explicit use of the shared caches present in almost all multicore chips today. Additional spatial blocking is required to maintain high performance even for large problem sizes. The largest performance gains are achieved on highly bandwidth-starved systems where many cores share a common path to memory, or where memory bandwidth does not scale across multiple sockets. Big shared caches are beneficial because they allow for reasonably large block sizes, reducing the inevitable overhead for spatial blocking. Note that we have so far not used any in-cache optimizations like outer loop unrolling or data alignment which we expect to yield additional performance improvements.

In view of the fact that Moore's Law will most probably hold for another decade, and that additional transistors will be used for implementing more cores and more cache per chip alike, our multicore-aware approach to temporal blocking is well suited for future processor designs.

Acknowledgment

Fruitful discussions with Johannes Habich, Jan Treibig, Markus Stürmer and Ulrich Rüde are gratefully acknowledged. We are indebted to Intel Germany for providing test systems and early access hardware for benchmarking. This work was carried out within the Bavarian framework of KONWIHR-II with additional financial support from BMBF through the project SKALB (grant 01IH08003A).

References

[1] B. Bergen, F. Hülsemann, U. Rüde: *Is 1.7×10^{10} Unknowns the Largest Finite Element System that Can Be Solved Today?* In: ACM/IEEE (Ed.): Proceedings of the ACM/IEEE SC 2005 Conference (Supercomputing Conference '05, Seattle, Nov 12–18, 2005).

[2] K. Datta, M. Murphy, V. Volkov, S. Williams, J. Carter, L. Oliker, D. Patterson, J. Shalf, K. Yelick: *Stencil Computation Optimization and Auto-tuning on State-of-the-Art Multicore Architectures.* In: ACM/IEEE (Ed.): Proceedings of the ACM/IEEE SC 2008 Conference (Supercomputing Conference '08, Austin, TX, Nov 15–21, 2008).

[3] M. Kowarschik: *Data Locality Optimizations for Iterative Numerical Algorithms and Cellular Automata on Hierarchical Memory Architectures.* PhD thesis, July 2004, SCS Publishing House, Germany. ISBN 3-936150-39-7.

[4] M. Frigo, C.E. Leiserson, H. Prokop, S. Ramachandran: *Cache-Oblivious Algorithms.* In: 40th Annual Symposium on Foundations of Computer Science, FOCS 99, Oct 17–18, 1999, New York, NY.

[5] T. Zeiser, G. Wellein, A. Nitsure, K. Iglberger, U. Rüde, G. Hager: *Introducing a parallel cache oblivious blocking approach for the lattice Boltzmann method.* Progress in CFD, vol. 8, No. 1–4, pp. 179–188, 2008.

[6] D.J. Kerbyson, A. Hoisie: *Analysis of Wavefront Algorithms on Large-scale Two-level Heterogeneous Processing Systems.* In Proc. Workshop on Unique Chips and Systems (UCAS2), IEEE Int. Symposium on Performance Analysis of Systems and Software (ISPASS), Austin, TX, 2006.

[7] J.D. McCalpin: *STREAM: Sustainable Memory Bandwidth in High Performance Computers.* http://www.cs.virginia.edu/stream/

[8] K.J. Barker, K. Davis, A. Hoisie, D.J. Kerbyson, M. Lang, S. Pakin, J.C. Sancho: *A Performance Evaluation of the Nehalem Quad-Core Processor for Scientific Computing.* Parallel Processing Letters **18**, No. 4, pp. 453–469 (2008).

[9] M. Meier: *Thread pinning by overloading pthread_create().* http://www.mulder.franken.de/workstuff/pthread-overload.c

[10] J. Treibig and G. Hager: *Introducing a Performance Model for Bandwidth Limited Loop Kernels*. Submitted to Workshop on Memory Issues on Multi- and Manycore Platforms, PPAM2009.