3D Finite Differences on Multi-core Processors
Abstract

This case study presents different optimization techniques for the implementation of multi dimensional finite difference stencils. In-core optimizations; data layout and approaches to improve data access performance are discussed. Domain decomposition with NUMA-aware data placement is covered. Finally usage examples of a few parallel programming models are presented. Here we explore Intel software tools features that range from hardware vector register programming support up to multiple parallel programming models.
1 Introduction

Finite Difference stencils typically arise in iterative finite-difference techniques employed to solve Partial Differential Equations (PDEs). For a given point in a regular grid, the stencil computation is a well-defined weighted contribution from a subset of neighbor points in both time and space. A 3-dimensional space stencil of order $O$ applied to the 3-dimensional array $v[i,j,k] = v_{i,j,k}$ can be defined as:

$$u_{i,j,k} = \sum_{\ell=1,\ldots,O/2} \left( c_{\ell,i-\ell,j,k} v_{i-\ell,j,k} + c_{\ell,i+\ell,j,k} v_{i+\ell,j,k} + c_{\ell,i,j-\ell,k} v_{i,j-\ell,k} + c_{\ell,i,j+\ell,k} v_{i,j+\ell,k} + c_{\ell,i,j,k-\ell} v_{i,j,k-\ell} + c_{\ell,i,j,k+\ell} v_{i,j,k+\ell} \right)$$

The same stencil is also called a $(3*O+1)$-point stencil. In the following we discuss optimization techniques using a 3-dimensional 8th-order Finite Difference (FD) stencils.
with symmetric-constant coefficients $c_l = c_{l, i, j, k}$ for $i,j,k=1,\ldots,O/2$. We assume single-precision data-types but most of the results can be extended to more generic cases and other stencil formats.

Graphically the (3*8+1)=25-point stencil can be represented as in Figure 1. Computationally it can be summarized as triply nested loops in $i,j,k$ updating array $U$ based on the values in array $V$ and coefficients array $C$:

$$U[i,j,k] = C[0]*V[i,j,k];$$
$$\text{for} \ (r=1; r<4; r++)$$
$$\quad U[i,j,k] += C[r]*(V[i-r,j,k]+V[i,j-r,k]+V[i,j,k-r]+V[i+r,j,k]+V[i,j+r,k]+V[i,j,k+r]);$$

![Figure 1: 25-point stencil. A new value for the center cell (orange) is obtained by weighted sum of the values in all the neighbor cells (blue).](image)

Typically the data structures (arrays) where the stencils are applied are much larger than the CPU data caches. With naïve implementations data reuse is limited by accessing only immediate neighbors $V[i\pm r, j, k], r< O/2$, in the fast (unit-stride) direction defined by $i \pm r$. Then reuse may not happen for elements that are already in the cache but are outside the stencil geometry. Poor data reuse creates unnecessary access requests to main memory which can result in performance degradation.

FD stencil computations can be implemented to take advantage of Intel® Xeon® processor family features like NUMA, memory hierarchy and SIMD instructions.

**The Baseline Code:** In what follows we consider the simpler case of a generic 8th-order 3D-FD computation from array $v$ to array $u$ of dimension $dx*dy*dz$, which schematically is:
2 In-core optimizations:

2.1 Introduction

Intel® Streaming SIMD Extensions (Intel® SSE) enhance the performance of computational intensive applications that execute the same operation on distinct elements in a data set. In our single-precision case, the improvement is obtained by performing single instruction multiple data (SIMD) parallelism with operations applied to four packed single-precision floating-point values in the same clock cycle. In the stencil in Figure 1 the X-direction is the only one where elements are contiguous in memory. Direction Y has array stride $dx$ and direction Z has stride $dx*dy$ as shown in the baseline code in Figure 2. This means that each floating-point $v[ix±k*dx]$ and $v[ix±k*dx*dy]$, $k=1,...,4$, belongs to distinct cache lines. Without SIMD vectorization, a single stencil computation underutilizes $4*4=16$ cache lines by accessing only one floating-point value from each of these cache-lines. Figure 3 is a graphical representation of 19 SSE registers being used to compute 4 stencils simultaneously. At least four floating-point elements are loaded from each cache-line, and twelve floating point elements belong to the cache-line(s) holding the X-direction elements.

```c
for(int iz=4; iz<dz-4; iz++) // 8th order 3D stencil, Z-direction */
for(int iy=4; iy<dy-4; iy++) { // Y-direction */
  v = &V[iy*dx + iz*dx*dy];
  u = &U[iy*dx + iz*dx*dy];
  /* x-direction (unit stride) */
  for(int ix=4; ix<dx-4; ix++) { /* X-direction */
    u[ix] = c0* v[ix]
                +c3*(v[ix+3]+v[ix-3]+v[ix+3*dx]+v[ix-3*dx]+v[ix+3*dx*dy]+v[ix-3*dx*dy])
  }
}
```

Figure 2: Our baseline 3D Finite Difference code for 8th order and symmetric constant coefficients.
Four stencils are computed simultaneously with register blocking in Intel® SSE.

The above estimate for the number of SIMD registers required on a fully vectorized stencil computation leads us to consider code changes. Loop splitting is an optimization that breaks a loop into two or more loops with fewer computations to reduce register pressure in the code: by splitting the stencil computations on independent \( x \), \( y \) and \( z \)-direction loops, the number of SIMD registers required to fully parallelize each new loop is considerably smaller than in the original loop. Figure 4 is an example of loop splitting. Intel® compilers provide a pragma/directive to disable fusion optimization on a loop-by-loop basis: \#pragma nofusion.

2.2 Compiler switches

Intel® compilers provide the easiest and fastest way to take advantage of Intel® SSE by supporting automatic vectorization methods (1). The first step is to choose the proper compiler switches for vectorization. In our example we have added the compiler switches

\[-O3 -xHOST -ip -ansi-alias -fno-alias -openmp -vec-report3\]

where the group “\(-O3 -xHOST\)” enables code generation for Intel® SSE extensions compatible with the processor model on the system in which the code was compiled. The switch “\(-vec-report3\)” enables compiler diagnostic information on vectorized and non-vectorized loops, including prohibiting data dependence information. This information can be used to identify code changes that enable further vectorization. Intel compilers also support guided auto-parallelization (GAP) tool that offers automated advice on adding compiler options, loop level pragmas, and source code changes to improve code vectorization (2). In our case, switches “\(-ansi-alias -fno-alias\)” indicate that there are
no aliased pointers in our particular C/C++ code and vectorization is safe. An alternative would be to use C99 restrict type qualifier in the function definitions. The switch “-fp” allows inter-procedural optimizations where it is possible. For more compiler optimization options see (3), for example.

Additionally compiler hints can be used to improve automatic intra-register vectorization and memory access behavior, for example, by inserting #pragma ivdep before the computational loops. It will overwrite any conservative data dependences that the compiler optimizer may have assumed. This can potentially enable additional loop vectorization.

```c
for(int iz=4; iz<dz-4; iz++)
    for(int iy=4; iy<dy-4; iy++) {
        v = &V[iy*dx + iz*dx*dy];
        u = &U[iy*dx + iz*dx*dy];
        /* x-direction (unit stride) split */
        #pragma ivdep
        for(int ix=4; ix<dx-4; ix++) {
            u[ix] = c0* v[ix]
                + c1* (v[ix+1] + v[ix-1])
                + c2* (v[ix+2] + v[ix-2])
                + c3* (v[ix+3] + v[ix-3])
                + c4* (v[ix+4] + v[ix-4]);
        }
        /* y-direction (dx stride) split */
        #pragma ivdep
        for (int ix= 4; ix<dx-4; ix++) {
            u[ix] += c1* (v[ix+dx] + v[ix-dx])
                + c2* (v[ix+2*dx] + v[ix-2*dx])
                + c3* (v[ix+3*dx] + v[ix-3*dx])
                + c4* (v[ix+4*dx] + v[ix-4*dx]);
        }
        /* z-direction (dx*dy stride) split */
        #pragma ivdep
        for (int ix= 4; ix<dx-dy; ix++) {
            u[ix] += c1* (v[ix+dx*dy] + v[ix-dx*dy])
                + c2* (v[ix+2*dx*dy] + v[ix-2*dx*dy])
                + c3* (v[ix+3*dx*dy] + v[ix-3*dx*dy])
                + c4* (v[ix+4*dx*dy] + v[ix-4*dx*dy]);
        }
    }
```

*Figure 4: Loop split to reduce register pressure in Intel® SSE vectorization.*

### 2.3 Intel® SSE/AVX Programming

A more aggressive approach is to program vectorized loops directly using Intel® SSE instructions. There are multiple ways to accomplish this. The most labor intensive way is to program directly in assembler. Alternatively, the compiler assembler intrinsic support makes Intel® SSE coding easier and provides a straightforward way to combine hardware level Intel® SSE instructions with high level C/C++ instructions in the same source code (4). Other easier and more portable alternatives are Intel C++ classes for
SIMD and the new CEAN extensions. Intel C++ classes for SIMD operate on arrays, or vectors of data, in parallel. Integer vector (Ivec) classes and floating-point vector (Fvec) classes are supported. They are named based on the underlying type of operation: F32vec4 operate on SSE 4-packed 32bit floating-point data, for example. Similarly, F32vec8 operate on AVX 8-packed 32bit floating-point data. Standard operators are +,+=,-,=,*=,/=,=.

Examples of advanced operators are Square Root, Reciprocal, and Reciprocal Square Root. It is worth noting that in the Nehalem and Westmere processors the penalty of using unaligned SIMD loads and stores is negligible compared to their aligned counterparts. But it is recommended that one enforce 16-byte aligned loads and stores where it is possible. On Sandy Bridge/AVX 32-byte alignment is recommended when possible (5). In our code example we have added preamble loops that allow the main loop to issue Intel® SSE instructions from aligned addresses of arrays v and u. Figure 5 exemplifies the use of the F32vec4 class to implement Intel® SSE vectorization in the y-direction loop. The integer value align_offset is the array index obtained from a preamble loop that aligns the array pointers to a 16-byte boundary. Consequently, vec_v and vec_u are pointers to the aligned memory address v+align_offset and u+align_offset, respectively. Advancing one array element in vec_v is equivalent to advance four elements in array v: the loop trip count vec_loop_trip and stride vec_dx are 4 times smaller than counterparts dx-4 and dx in the original scalar version. For each constant coefficient ci the constructor F32vec4 packs four identical copies of ci into vec_ci.

#include <fvec.h>

// Coefficients in SSE register form
const F32vec4 vec_c1(c1), vec_c2(c2), vec_c3(c3), vec_c4(c4);

// Aligned SSE arrays
F32vec4* vec_v = (F32vec4*)(v+align_offset);
F32vec4* vec_u = (F32vec4*)(u+align_offset);

// loop on SSE registers
__assume_aligned(vec_u, 16); __assume_aligned(vec_v, 16);
for(int ix=0; ix<vec_loop_trip; ix++) {
    vec_u[ix] += vec_c1* (vec_v[ix+vec_dx] + vec_v[ix-vec_dx])
    + vec_c2* (vec_v[ix+2*vec_dx] + vec_v[ix-2*vec_dx])
    + vec_c3* (vec_v[ix+3*vec_dx] + vec_v[ix-3*vec_dx])
    + vec_c4* (vec_v[ix+4*vec_dx] + vec_v[ix-4*vec_dx]);
}

// Scalar post-loop (remaining elements)
for(int ix=4* vec_loop_trip; ix<dx-4; ix++) {
    u[ix] += c1* (v[ix+dx] + v[ix-dx])
    + c2* (v[ix+2*dx] + v[ix-2*dx])
    + c3* (v[ix+3*dx] + v[ix-3*dx])
    + c4* (v[ix+4*dx] + v[ix-4*dx]);
}

Figure 5: Using C++ classes to implement Intel® SSE vectorized loops. Each arithmetic operation within the Intel® SSE register loop is equivalent to four single precision scalar operations. Similar AVX code with F32vec8 class can compute eight single precision scalar operations.
These classes also provide a path to maintain assembler level code that can be easily transitioned to new hardware language extensions like Intel® Advanced Vector Extensions (Intel® AVX) (6). Intel® AVX is a new instruction set extension to Intel® SSE that is part of the upcoming Sandy Bridge processor family. Intel® AVX improves performance due to wider vectors, new extensible syntax, and rich functionality. The new 256 bit instruction set has SIMD registers that are twice wider than the Intel® SSE 128 bit instruction set. In practice it means the registers in Figure 3 are twice wider in Intel® AVX. Current Intel® compilers already support code generation and C++ classes for Intel® AVX SIMD. Programmers still can rely on the Intel® C/C++/Fortran compiler’s auto-vectorizers to generate Intel® AVX vector loops. Programmers also have the option to update the 3D-FD code in Figure 5 to operate with 8 packed 32bit floating-point numbers per cycle by simply moving from F32vec4 to F32vec8 classes and enforcing 32-byte alignment. These Intel® AVX classes are also defined in the include files distributed with Intel® compilers.

One additional alternative implementation for the above explicit vectorization is to take advantage of Intel® Cilk™ Plus extensions for array notations (7). These C/C++ extensions provide a direct way to express array sections directly in the source code using the syntax

\[
\text{section}\_\text{operator} := \left[\text{lower bound} : \text{length} : \text{stride}\right]
\]

An explicitly Extension for Array Notation-based vectorized AVX version of the code in Figure 5 can be written as in Figure 6. In this example we also demonstrate how to use __assume_aligned() and __assume() declarations to hint the Intel compiler on data alignment information. Declarations __assume_aligned(u,32) and __assume_aligned(v,32) indicate that bases addresses of arrays u and v are 32byte aligned. Declarations of the type __assume(<var> %8 == 0) indicates that the integer scalar <var> is a multiple of 8 and, consequently, if the address of v[ix] is 32byte aligned, then the address of v[ix+ <var>] the offset is also 32byte aligned. Note that the use of __assume() and __assume_aligned() is not required: they are a resource to minimize compiler code generation by asserting known alignment properties.
// Assume u and v[k*dx], k=-4,...,4, are 32byte aligned arrays
__assume_aligned(u,32);
__assume_aligned(v,32);

// 8 single precision floats on an AVX register
int dx_1=dx; __assume(dx_1%8 == 0);
int dx_2=2*dx; __assume (dx_2%8 == 0);
int dx_3=3*dx; __assume (dx_3%8 == 0);
int dx_4=4*dx; __assume (dx_4%8 == 0);

// loop using AVX registers (8 single precision floats)
#pragma simd
for(int ix=0; ix<vec_loop_trip; ix+=8) {
    u[ix:8] += c1* (v[ix+dx_1:8] + v[ix-dx_1:8])
               + c2* (v[ix+dx_2:8] + v[ix-dx_2:8])
               + c3* (v[ix+dx_3:8] + v[ix-dx_3:8])
               + c4* (v[ix+dx_4:8] + v[ix-dx_4:8]);
}

Figure 6: Using Cilk Array Notation to implement Intel® AVX vectorized loops. Each arithmetic operation within the Intel® AVX register loop is equivalent to eight single precision scalar operations. Although not required, alignment assumption can be given to the compiler to minimize code generation.

3 Data layout & Optimizations for data access

3.1 Data Layout
PDEs solved using finite difference schemes typically require many 3-dimensional arrays of data to be accessed to compute each given output value $u_{i,j,k}$: at least the original input array $v_{i,j,k}$ plus its neighbor values $v_{i+1,j,m,k}$ in the stencil combined with the proper coefficients $c_{i+1,j,m,k}$ defined by the physics of the problem. Data is usually stored in contiguous arrays of floating points. This layout also allows for vectorization. The drawback is that computations dealing with multiple arrays will require coding that improves data locality.

3.2 Cache Blocking
The active data set in HPC applications is typically orders of magnitudes larger than the total amount of processor caches. Data blocking can be used to increase spatial locality and data reuse: data should be partitioned based on the number of cores/threads available in the systems and on the size of data-caches. Data blocking can be implemented in many ways and at multiple levels. Examples are core blocking and cache blocking. If Simultaneous MultiThreading (SMT) is enabled there are two threads to every physical core. In this case, thread blocking and core blocking may be implemented differently.

On Figure 7, cache blocking is implemented by accessing the original $dx \times dy \times dz$ data arrays in sub-blocks of size $CX \times CY \times CZ$. The goal is to define a cache blocking size small enough to minimize last level cache data misses and large enough to maximize data reuse. Additionally, thread blocking can be adopted to further divide cache blocks into smaller sub-blocks (thread blocks) and schedule threads within the processor core in
which the cache block belongs to. In our implementation we treated cache blocking and thread blocking as the same. Register blocking will be addressed later with SIMD data parallelism. A $CX \times CY \times CZ$ cache blocking scheme is represented in Figure 7.

```c
for(int bz=zbegin; bz<zend; bz+=CZ)
    for(int by=ybegin; by<yend; by+=CY)
        for(int bx=xbegin; bx<xend; bx+=CX)
            for(int iz=bz; iz<MIN(bz+CZ,zend); iz++)
                for(int iy=by; iy<MIN(by+CY,yend); iy++)
                    for(int ix=bx; ix<MIN(bx+CX,xend); ix++)
                        /* compute stencil */
```

Figure 7: Outline of a 3 dimensional blocking. $CX$, $CY$ and $CZ$ are the blocking constants that are based on the cache structure of the hardware.

From a performance perspective, blocking in the unit stride first-dimension $x$ has to be large enough to allow the inner loop to take advantage of vectorization.

Cacheability control allows programmers to apply knowledge of the program data flow (5). In particular, Streaming Non-temporal Stores can potentially increase stores bandwidth by writing around the processor caches, therefore minimizing cache pollution due to data that will not be accessed again in the near future. If the 64 bytes that fit within a cache line are written consecutively, stream stores may be used. By default, Intel® compilers automatically determine whether a streaming store should be used for each array. Programmers can explicitly define `VECTOR NONTEMPORAL` pragma/directive that directs the compiler to use non-temporal (streaming) stores as shown in Figure 8.

```c
#pragma vector nontemporal (u)
#pragma ivdep
for(int ix=xbegin; ix<xend; ix++) {
}
```

Figure 8: Pragma used to hint the non-temporal (streaming) nature of the store operation for the array $u$.

For additional information on the architecture and programming environment of the Intel® 64 and IA-32 processors refer to (8)

4 Problem Decomposition and NUMA-aware Data Placement:

Current multiprocessor systems are based in NonUniform Memory Access (NUMA) architecture. A CPU can access its own local memory or remote memory physically located in another CPU socket, that is, physical memory located in another NUMA node. The total shared memory available is the aggregated amount of memory for all NUMA nodes on the system. Locality matters: access latency to remote NUMA node memory is typically higher than latency to local NUMA memory. Conversely, memory bandwidth in local NUMA node can be considerably greater than memory bandwidth to remote
NUMA nodes. This makes desirable to have threads accessing memory in the same NUMA node the thread is running.

The way to guarantee data affinity to each CPU depends on the parallelization model in use:

- In \textit{process-parallel execution}, HPC applications are typically implemented using MPI (Message Passing Interface). MPI implementations that support process affinity pinning will maximize local NUMA memory access by keeping each MPI process bound to a given CPU socket and by allocating its data in physical memory located in the same socket. Intel® MPI provides process affinity controls to pin a particular MPI process to a corresponding CPU and to avoid undesired process migration (9).

- In \textit{multiprocessor shared-memory threading}, the application consists of a single process that is parallelized using threads. Wherever possible, the programmer should partition data structures to have data placed in the same CPU socket which runs the threads consuming that data. These threads should have their affinity set to the CPU in the same NUMA-node to avoid unwanted OS scheduled thread migration. To ensure that the threads can allocate their required memory on a specific NUMA-node, both Linux* and Windows* provide means to control the allocation behavior. On Linux, \texttt{malloc} simply reserves the memory. The OS actually assigns the physical page only when data is touched for the first time (first touch policy) (10). This default policy works well in many cases, assuming the programmer takes advantage of the policy: the data can be initialized (first touch) with a parallel routine resembling the way data will be accessed by threads in the processing phase. Thus, memory pages are correctly pinned to the CPU socket that contains the threads accessing these pages. For details on Windows* see (11) and (12), for example.

- In a \textit{hybrid MPI with shared-memory threading} model, one can combine the above approaches. At the highest level MPI processes are used to parallelize the work across the number of CPU sockets available in the system. Each MPI process is then parallelized via shared-memory threading based on the number of cores available in the CPU socket. This can potentially combine advantages of MPI affinity pinning across NUMA nodes with shared memory threading within each NUMA node.

Below we compare two implementations: shared-memory threading only, and hybrid MPI together with shared-memory threading on a system with two NUMA nodes. Both implementations were parallelized by splitting the work defined in the slowest direction $Z$ outer loop. In the pure shared-memory threading implementation each thread performed computations on a contiguous $[dx, dy, dz/nt]$ section of data, where $nt$ is the total number of threads. We ran two variants of this threading example: \textit{Threading Baseline} is a straightforward OpenMP implementation; and \textit{NUMA-aware Threading} is a variant of the same code where a parallel data initialization routine was added to replicated the data access pattern of the way the $z$-loop was parallelized.

In the \textit{Hybrid MPI+Threading} version data was split into two contiguous $[dx, dy, dz/2]$ sections. Each section of data was assigned to a MPI processes. Each MPI process was
then threaded with \( nt/2 \) threads. Figure 9 compares performance improvement with relation to the Threading Baseline implementation of this example. This synthetic 480x480x400 example of the 3DFD code ran in a dual socket X5680 Westmere system. The problem size was chosen small enough so that NUMA effects can be easily noticeable. The NUMA-aware Threading version of the code showed a 1.4X performance improvement when thread affinity was properly set using the KMP_AFFINITY runtime affinity interface in the OpenMP library of the Intel Compiler (13). The Hybrid MPI+Threading version guaranteed further data locality by allocating the data structures on their respective NUMA nodes and showed a 1.7X performance improvement*. 

![Figure 9: Effect of NUMA memory affinity on performance. Implementations that are sensitive to local versus remote NUMA memory bandwidth can take advantage of local memory affinity. The bars represent the performance improvement with relation to the non NUMA-aware implementation of this example.](image)

In the following paragraphs we discuss optimizations within a processing node of a cluster.

5 Parallelization in Shared Memory:

This section is a brief overview of multiple approaches supported by the Intel software tools to develop threaded applications. Using a single sample case, this section exemplifies OpenMP, TBB (Threading Building Block) and Cilk Plus based implementations. It will not cover parallelization based on either Posix threads or MPI (Message Passing Interface) – although both techniques are also supported by the Intel tools.

As previously stated, data blocking can be used to increase spatial locality and data reuse. In this example, data is partitioned across cores/threads available in the system by defining thread block sizes \( TX, TY, \) and \( TZ \) following the notation in (14). This can be
accomplished in multiple ways: for example, by either using threading model constructs that support blocking, or by defining a list of threading blocks to be assigned to the threads. To simplify the discussion, we consider no blocking in the unit stride vectorizable first-dimension $x$ in the case that follows. That is, $TX=dx$.

Assuming vectorization in the unit stride first-dimension $x$ and only thread blocking $TY$ and $TZ$ in the second and third dimensions $y$-$z$, respectively, an OpenMP `parallel for` loop can be used to issue the threading blocks. In fact, both loops in $z$ and $y$ can be collapsed if the number of threading blocks along the third dimension $z$ is not high enough to keep all available threads busy. This case is presented in Figure 10. The advantage of an OpenMP implementation over Posix threads is that there is no need to maintain a user managed thread pool: the Intel OpenMP library automatically maintains a thread pool to avoid thread creation overheads.

```c
#pragma omp parallel for collapse(2)
for(int bz=zbegin; bz<zend; bz+=TZ)
    for(int by=ybegin; by<yend; by+=TY)
        for(int iz=bz; iz<MIN(bz+CZ,zend); iz++)
            for(int iy=by; iy<MIN(by+CY,yend); iy++)
                for(int ix=xbegin; ix<xend; ix++)
                    /* compute stencil */

Figure 10: OpenMP provides a straightforward way to assign threading blocks to threads. In this case, the two outer loops are collapsed in terms of OpenMP parallelization.
```

In our reference implementation we adopt the approach of defining a list of threading blocks to be assigned to the threads as shown in Figure 11. First, this allows the programmer to define any the order in which the threading blocks are assigned to threads; second this allow the same example framework to be used in other threading models.

```c
struct block_struct{
    int iyb;
    int izb;
};

block_struct blocking[NUM_BLOCKS+1];
int index=0;
for (int izb= zbegin; izb<zend; izb+=TZ)
    for(int iyb= ybegin; iyb<yend; iyb+=TY) {
        blocking[index].iyb = iyb;
        blocking[index].izb = izb;
        index++;
    }

Figure 11: Defining a list of threading blocks to be assigned to the threads. Obviously the other of the list can be changed by rearranging the loops in red-and-black, reverse-Y-Z, or other ordering schemes.
```

The OpenMP code excerpts in Figure 12 are examples of two possible implementations based on a list of threading blocks. The first variant uses an OpenMP `parallel for` pragma to assign threading blocks to the threads. Note that `num_blocks` can be larger than the number of threads available in the system: the goal is to improve data locality and achieve data granularity that allows for load balance. The pseudo function `3DFD_BLK`
computes a vectorized finite difference in the threading block defined by [0..dx, iyb..iyb+TY, izb..izb+TZ]. Ellipses "..." in the function call and in the following examples represent additional parameters that might be required, like the stencil coefficients, for example. The second variant assigns threading blocks via parallel OpenMP task pragmas.

/* Variant 1 */
#pragma omp parallel for schedule(dynamic)
for (int i=0; i<num_blocks; i++) {
    int iyb = blocking[i].iyb;
    int izb = blocking[i].izb;
    3DFD_BLK(V, U, dx, dy, dz, iyb, izb, TY, TZ, ...);
}

/* Variant 2 */
#pragma omp parallel schedule(dynamic)
{
    #pragma omp single
    {
        #pragma omp task untied
        for (int i=0; i<num_blocks; i++) {
            int iyb = blocking[i].iyb;
            int izb = blocking[i].izb;
            #pragma omp task
            3DFD_BLK(V, U, dx, dy, dz, iyb, izb, TY, TZ, ...);
        }
    }
}

Figure 12: Two variants of OpenMP implementation that uses a list of threads to be dispatched. The first variant uses an OpenMP parallel for pragma. The second implementation uses OpenMP tasks. The pseudo function 3DFD_BLK computes the vectorized finite difference in the threading block defined by [0..dx, iyb..iyb+TY, izb..izb+TZ].

OS scheduled thread migration can affect performance of OpenMP based implementations. Using the KMP_AFFINITY runtime affinity interface (13) one can take control of thread placement. In our example, OpenMP dynamic scheduling together with KMP_LIBRARY set to “turnaround” can be used to increase load balance on high core count systems.

In similar fashion, parallelization can be implemented using Intel® compilers Cilk™ Plus extensions. Like in OpenMP, a first variant can be implemented using a cilk_for loop as shown in Figure 13. An alternative implementation could be based on cilk_spawn.

cilk_for (int i=0; i<num_blocks; i++) {
    int iyb = blocking[i].iyb;
    int izb = blocking[i].izb;
    3DFD_BLK(V, U, dx, dy, dz, iyb, izb, TY, TZ, ...);
}

Figure 13: Two variants of Cilk++ implementation. Similarly to OpenMP, the first variant is implemented with a cilk_for, and the second implementation uses cilk_spawn.

Although Intel TBB (Threading Building Blocks) can also be used to dispatch the threading blocks defined in the list blocking, in Figure 14 we present a variant that takes advantage of TBB’s blocked_range2d template class (15). A parallel_for loop is used as a load-balanced way to apply the function 3DFD_BLK to each element of the loop. The loop
partitioning is defined by a two-dimensional interval \( [y_{\text{begin}}, y_{\text{end}}) \times [z_{\text{begin}}, z_{\text{end}}) \), with maximum block sizes \( TY \) and \( TZ \), respectively.

```cpp
class __declspec(target(mic)) tbb_parallel_task {
    float *const my_V;
    float *const my_U;
    int const my_dx;
    int const my_dy;
    int const my_dz;
    ...
	public:

    void operator() (const tbb::blocked_range2d<int> &r) const {
        3DFD_Blk(my_V, my_U, my_dx, my_dy, my_dz,
            r.rows().begin(), r.cols().begin(), // replaces iyb, iyz
            r.rows().end()-r.rows().begin(), // replaces TY
            r.cols().end()-r.cols().begin(), // replaces TZ
            ...);
    }

tbb_parallel_task (float V[], float U[], int dimx, int dimy, int dimz, ...) :
    my_V(V),
    my_U(U),
    my_dx(dx),
    my_dy(dy),
    my_dz(dz),
    ...
{
}

};

// main program:
// defined 2D partition based on Thread Blocking TY, TZ
    tbb::parallel_for (tbb::blocked_range2d<int> (ybegin, yend, TY, zbegin, zend, TZ),
        tbb_parallel_task (V, U, dx, dy, dz, ...));
```

Figure 14: A TBB parallel_for loop uses a blocked_range2d class to automatically partition the work among threads. On this example, the two-dimensional interval \( [y_{\text{begin}}, y_{\text{end}}) \times [z_{\text{begin}}, z_{\text{end}}) \), is split with maximum block sizes \( TY \) and \( TZ \), respectively.

Refer to (16) for a collection of technical papers to provide software developers with technical information on application threading, synchronization, memory management and programming tools.
6 Works Cited
About the Authors

Leonardo Borges is a Sr. Application Engineer based in Houston, TX, enabling HPC in the Energy vertical. His background is in numerical analysis and parallel computing. Leo received his Ph.D. in Computer Science from Texas A&M University.

Philippe Thierry is a Sr. Staff Engineer in the Energy Application Engineering Team at Intel in Paris, and specializes in HPC and seismic imaging. Philippe received his Ph.D. in Geophysics from the Paris School of Mines and University of Paris VII.

Notices
INFORMATION IN THIS DOCUMENT IS PROVIDED IN CONNECTION WITH INTEL PRODUCTS. NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS DOCUMENT. EXCEPT AS PROVIDED IN INTEL’S TERMS AND CONDITIONS OF SALE FOR SUCH PRODUCTS, INTEL ASSUMES NO LIABILITY WHATSOEVER AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO SALE AND/OR USE OF INTEL PRODUCTS INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT.

UNLESS OTHERWISE AGREED IN WRITING BY INTEL, THE INTEL PRODUCTS ARE NOT DESIGNED NOR INTENDED FOR ANY APPLICATION IN WHICH THE FAILURE OF THE INTEL PRODUCT COULD CREATE A SITUATION WHERE PERSONAL INJURY OR DEATH MAY OCCUR.

Intel may make changes to specifications and product descriptions at any time, without notice. Designers must not rely on the absence or characteristics of any features or instructions marked "reserved" or "undefined." Intel reserves these for future definition and shall have no responsibility whatsoever for conflicts or incompatibilities arising from future changes to them. The information here is subject to change without notice. Do not finalize a design with this information.

The products described in this document may contain design defects or errors known as errata which may cause the product to deviate from published specifications. Current characterized errata are available on request.

Contact your local Intel sales office or your distributor to obtain the latest specifications and before placing your product order.

Copies of documents which have an order number and are referenced in this document, or other Intel literature, may be obtained by calling 1-800-548-4725, or go to: http://www.intel.com/design/literature.htm

Intel, the Intel logo, and Xeon are trademarks of Intel Corporation in the U.S. and other countries.

Copyright© 2001 Intel Corporation. All rights reserved.

*Other names and brands may be claimed as the property of others.

For more complete information about performance and benchmark results, visit www.intel.com/benchmarks
## Optimization Notice

Intel® compilers, associated libraries and associated development tools may include or utilize options that optimize for instruction sets that are available in both Intel® and non-Intel microprocessors (for example SIMD instruction sets), but do not optimize equally for non-Intel microprocessors. In addition, certain compiler options for Intel compilers, including some that are not specific to Intel micro-architecture, are reserved for Intel microprocessors. For a detailed description of Intel compiler options, including the instruction sets and specific microprocessors they implicate, please refer to the “Intel® Compiler User and Reference Guides” under “Compiler Options.” Many library routines that are part of Intel® compiler products are more highly optimized for Intel microprocessors than for other microprocessors. While the compilers and libraries in Intel® compiler products offer optimizations for both Intel and Intel-compatible microprocessors, depending on the options you select, your code and other factors, you likely will get extra performance on Intel microprocessors.

Intel® compilers, associated libraries and associated development tools may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include Intel® Streaming SIMD Extensions 2 (Intel® SSE2), Intel® Streaming SIMD Extensions 3 (Intel® SSE3), and Supplemental Streaming SIMD Extensions 3 (Intel® SSSE3) instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors.

While Intel believes our compilers and libraries are excellent choices to assist in obtaining the best performance on Intel® and non-Intel microprocessors, Intel recommends that you evaluate other compilers and libraries to determine which best meet your requirements. We hope to win your business by striving to offer the best performance of any compiler or library; please let us know if you find we do not.

Notice revision #20101101