Abstract

In this paper, we use our stencil code generation and auto-tuning framework Parus to optimize and parallelize the most compute intensive stencil calculations of an anelastic wave propagation code, which was used to conduct numerous significant simulations at the Southern California Earthquake Center. From a straight-forward specification of the stencil calculation, Parus automatically creates an implementation targeted at the chosen hardware platform and applies hardware-specific optimizations including cache blocking, loop unrolling, and explicit vectorization. We show that, using this approach, we are able to speed up individual compute kernels by a factor of $2.4 \times$ on average, and reduce the time required to compute one time step of the entire simulation by 47% in a weak and up to 129% in a strong thread scaling setting.

Keywords: wave propagation, finite differences, stencil computations, auto-tuning

1. Introduction

The finite difference method enjoys great popularity in practice. Not only are the underlying numerics simpler and thus are the methods easier to implement than, for instance, finite element methods, but they also map well to the hardware on which the codes are executed. The regular structure of finite difference codes, which apply a finite difference stencil to each grid point, results in regular data access patterns, which are favored by the hardware. The regularity of finite difference grids furthermore facilitates making best use of the memory hierarchy.

Nevertheless, in order to elicit the machine’s full compute power, both explicit parallelization and meticulous architecture- and application-specific tuning is required, as microarchitectures are becoming increasingly complex and concurrency is exposed to the programmer. Manually optimizing a code to target a specific hardware platform not only is both a time consuming and error-prone process, but also makes the code unmaintainable and non-portable to other architectures.

In this paper, we take the approach to generate and tune stencil codes automatically from a straight-forward stencil specification using our stencil code generating and auto-tuning framework Parus [1, 2] in order to increase code performance and tune the implementation to the target hardware platform. The generated implementations are then integrated into the full application.

We focus on the anelastic wave propagation code AWP-ODC of the Southern California Earthquake Center (SCCEC), which was developed by Olsen, Day, Cui, and Dalguer. It is a finite difference code for simulating both dynamic rupture and earthquake wave propagation. It has been used to conduct numerous significant simulations at...
the SCEC, including the simulation of a magnitude-8 earthquake rupturing the entire San Andreas Fault from central California to the Mexican border, a fault length of 545 km [3].

The governing elastodynamic equations of the model used in AWP-ODC is the following system of PDEs [3, 4]:

\[ \frac{\partial \mathbf{u}}{\partial t} = \rho^{-1} \nabla \cdot \mathbf{\sigma} \]  
\[ \frac{\partial \mathbf{\sigma}}{\partial t} = \lambda (\nabla \cdot \mathbf{u}) \mathbf{I} + \mu (\nabla \mathbf{u} + \nabla \mathbf{u}^T). \]

The dependent variables are the velocity vector field \( \mathbf{u} = (u_x, u_y, u_z) \) and the stress tensor \( \mathbf{\sigma} = (\sigma_{xx}, \sigma_{xy}, \sigma_{xz}, \sigma_{yx}, \sigma_{yy}, \sigma_{yz}, \sigma_{zx}, \sigma_{zy}, \sigma_{zz}) \). \( \lambda \) and \( \mu \) are the Lamé coefficients, \( \rho \) is the density, and \( \mathbf{I} \) is the identity tensor.

This paper is organized as follows. We start with a description of the \textsc{Parus} framework, then give some details on the AWP-ODC code and discuss how we integrate \textsc{Parus} into the toolchain. We proceed with a description of the target hardware platform, the tuning process of individual stencil kernels of AWP-ODC, and finally present some full-application scaling results.

2. \textsc{Parus}: A Framework for Generating and Auto-tuning Parallel Stencil Codes

The \textsc{Parus} framework is a code generation and auto-tuning tool for the class of stencil computations striving for both programmer productivity and performance. Using a small domain specific language (DSL), the user defines the stencil kernel using a C-like syntax. The code generation is driven by a Strategy: a specification of the parallelization and code optimization methods, formulated in another DSL. Strategies are designed to be independent both of the stencil specification and the hardware platform, thus emphasizing \textsc{Parus}'s productivity and portability aspects by separating the point-wise computation from the algorithmic implementation of the stencil kernel. Strategies incorporate domain-specific knowledge that enables optimizing the code beyond the abilities of current general purpose compilers. As the performance of stencil computations typically is limited by the available bandwidth to the memory subsystem because of their low arithmetic intensity, i.e., the low number of floating point operations per transferred data element, it is important to make efficient use of caches or scratch pad memory by optimizing spatial and temporal data locality.

Candidates for bandwidth-saving schemes include cache blocking techniques [5, 6] and methods to block across multiple time steps [7, 8, 9, 10], which effectively increase the arithmetic intensity. The benefit of cache blocking is improved temporal data locality, i.e., more efficient use of the cache, which results in a performance increase. By decomposing the grid into cache size-dependent small subdomains it is ensured that data loaded into the cache can be reused before being evicted due to capacity misses.

The idea is to encode such schemes in \textsc{Parus} Strategies. Complementary hardware-aware programming techniques such as NUMA-aware data initialization, software prefetching, or cache bypassing, which help to reduce bandwidth usage further, are implemented directly in the code generator. By adapting the hardware architecture specification and the code generation back-end, \textsc{Parus} is able to support future hardware microarchitectures and programming paradigms. Currently, traditional CPU architectures and NVIDIA CUDA-programmable GPUs are supported.

2.1. An Example Strategy

Strategies are typically parametrized. For instance, in a cache blocking Strategy, the cache block size has an impact on performance since the cache sizes vary among hardware platforms. Thus, the cache block size needs to be a tunable Strategy parameter. We use auto-tuning as the methodology used to determine the best parameter value, given a stencil kernel and the hardware platform on which the code is executed. Auto-tuning means conducting automated benchmarks, i.e., actually executing the code on the target platform and measuring its performance, and using some search method to explore the parameter space.

The idea behind Strategies is to provide a clean mechanism which separates the implementation of parallelization and bandwidth-optimizing techniques from the actual stencil computation. In this way, the implementation of the algorithm can be reused for other stencils.

Listing 1 shows a simple cache blocking Strategy. It is the one used for the performance studies discussed in Sections 6 and 7.
The Strategy iterates over all the time steps in the $t$ loop. Within one time step it iterates in blocks $v$ of size $cb$ over the “root domain” $u$, i.e., the entire domain to which to apply the stencil. Both the root domain and the size of the subdomain $v$ are given as Strategy parameters. The blocks $v$ are executed in parallel by virtue of the parallel keyword, which means that the subdomains $v$ are dealt out in a cyclic fashion to the worker threads. The parameter $chunk$ to the schedule keyword defines how many consecutive blocks one thread is given. In the $p$ loop, the stencil is applied for each point in the subdomain $v$.

The Strategy argument $cb$ has a specifier, auto, which means that this parameter will be interfaced with the auto-tuner: it is exposed as a command line parameter of the generated benchmarking harness so that the auto-tuner can provide values for $cb = (c_1, c_2, \ldots, c_d)$, where $d$ is the dimensionality of the stencil, and pick the one for which the best performance is measured.

### 2.2. The Framework’s Architecture

**Patux** is built from four core components as shown in the high-level overview in Fig. 1: the two parsers for the two input files — the stencil definition and the Strategy, — the actual code generator, and the auto-tuner.

The stencil specification parser creates an internal representation from the textual representation of the specification. The internal representation consists of the domain size and number-of-iterations attributes, and a graph representation of the actual point-wise computation, which can consist of multiple stencil expressions operating on one or multiple time steps of one or multiple grids. The Strategy parser transforms the Strategy code into an abstract syntax tree (AST) represented as Cetus-based IR objects. Cetus [11] is a source-to-source compiler infrastructure developed at Purdue University (USA). Both the stencil specification and the Strategy parsers were implemented using Coco/R [12], an open source parser generator developed at the University of Linz (Austria), which can generate recursive descent LL($k$) parsers.
The internal representation of the stencil and Strategy AST structures are passed to the code generator, along with an additional configuration describing the characteristics of the target hardware and the target programming model. The code generator produces C code for variants of the stencil kernel and also creates an initialization routine that implements a NUMA-aware data initialization based on the parallelization scheme used in the kernel routine.

The objective of the code generator is to translate the Strategy, into which the stencil was substituted, into the final C code. In particular, it transform Strategy loops into C loops and parallelizes them according to the specification in the Strategy. If desired, inner-most loops containing the stencil computation are unrolled and vectorized.

Along with an implementation for the stencil kernel, the code generator also creates a benchmarking harness from an architecture- and programming model-specific template into which the dynamic memory allocations, the grid initializations, and the kernel invocation are substituted.

Based on a range specification for the parameters and optional constraints (e.g., to assert that the number of v-blocks in Listing 1 is at least the number of running threads), the auto-tuner runs the benchmark executable repeatedly with varied parameter configurations. The exploration of the search space is driven by a derivative-free search method. Parus comes with a selection of methods, including exhaustive search, a greedy search searching along coordinate axes and fixing the best value before progressing to the next axis, general combined elimination [13], simplex search, and a genetic algorithm.

3. The AWP-ODC Code

The numerics of AWP-ODC is based on a finite difference discretization of the velocity-stress wave equations (Eq. (1)). They are solved with an explicit second order time integrator on a staggered grid with equidistant mesh points. In space, the discrete differential operators are fourth order discretizations of the respective operators. A split-node approach is used to model dynamic fault ruptures [4]. As absorbing boundary conditions, the code uses perfectly matched layers and dampening within a sponge layer.

The original code is a pure MPI code written in Fortran. It is written in such a way that a number of routines compute the inner grid points of the velocity vector field and of the stress tensor field, respectively. Therefore they are pure, homogeneous stencil computations, while other, dedicated routines take care of the boundary computation and the halo exchange. In this work, we concentrate on optimizing the few finite difference computation routines in which most of the compute time is spent. We rewrite the corresponding stencils as Parus stencil specifications and replace the original code with the code generated and tuned by Parus.

Table 1 shows the 7 routines in which most of the compute time is spent when the code is run in elastic mode. These are also the routines which we focus on with our optimization work. The routines with the same computation structures are shown in the same row. All the routines are pure stencil computations. The three routines uxx1, vyy1, wzz1 compute the components $u_x$, $u_y$, $u_z$ of the velocity vector field from the the stress. Conversely, each of the routines xy1, xx1, yz1 compute one of the stress tensor components $\sigma_{xy}$, $\sigma_{xz}$, $\sigma_{yz}$ from the velocity. xyz1 computes all the diagonal stress tensor components $\sigma_{xx}$, $\sigma_{yy}$, $\sigma_{zz}$ at once.

For a local domain size of $256^3$ grid points, 78% of the entire run time is spent in these 7 routines; the velocity computation routines uxx1, vyy1, wzz1 combined use around 32% of the time, while the single stress component computation routines xy1, xx1, yz1 use a share of 19%, and the diagonal components in xyz1 27%. The percentage numbers depend on the local domain size since for smaller local domains the surface-to-volume ratio increases and thus there is a larger overhead to compute and exchange the boundary grid points. For a $128^3$-sized local domain, the run time spent in the 7 routines is around 61%, for a $512^3$-sized domain it is 88%.

The table also shows the number of floating point operations and the arithmetic intensities, which allow for rough performance estimate.

4. Integrating Parus-generated Kernels into AWP-ODC

For better ease of use and integration with existing toolchains, Parus can act as a pre-processor which extracts annotated sections in a source file and interprets it as a Parus stencil specification. As output, it produces C implementations of the thus specified stencil computations as well as a new version of the source file with the annotated stencil specification sections replaced by calls to the generated kernel functions. This new version is then compiled in
Matthias Christen and Olaf Schenk / Procedia Computer Science 9 (2012) 956 – 965

Table 1: Routines of AWP-ODC in which most compute time is spent.

<table>
<thead>
<tr>
<th>Routine</th>
<th>Description</th>
<th>% Exec. Time</th>
<th>Flops/Stencil</th>
<th>Arith. Intensity</th>
</tr>
</thead>
<tbody>
<tr>
<td>uxx1, vyy1, wzz1</td>
<td>Velocity components $u_x$, $u_y$, $u_z$</td>
<td>32%</td>
<td>20</td>
<td>0.82 Flop/Byte</td>
</tr>
<tr>
<td>xy1, xz1, yz1</td>
<td>Stress tensor components $\sigma_{xy}$, $\sigma_{xz}$, $\sigma_{yz}$</td>
<td>19%</td>
<td>14</td>
<td>0.69 Flop/Byte</td>
</tr>
<tr>
<td>xyz1</td>
<td>Stress tensor components $\sigma_{xx}$, $\sigma_{yy}$, $\sigma_{zz}$</td>
<td>27%</td>
<td>74</td>
<td>1.65 Flop/Byte</td>
</tr>
</tbody>
</table>

the regular toolchain in lieu of the modified source. As Parus is built modularly, the front-end could be replaced by a parser that could detect stencil computations, e.g., in a Fortran source code. This might be addressed in future work, as it would allow using Parus less intrusively.

Listing 2 shows the Parus stencil specification of the stencil of the uxx1 routine, which was inserted into the original Fortran source delimited by \#pragma patus annotations. It is a one-to-one translation of the original Fortran code. In the stencil specification, the domainsize definition defines the rectangular iteration space. The identifiers \texttt{nxb}, \texttt{nxe}, etc. are the names of the loop bound variables in the original code. \texttt{t_max=1} tells Parus that only one time step is to be performed within the generated stencil kernel function.

```fortran
stencil uxx1 {
  domainsize = (nxb .. nxe, nyb .. nye, nzb .. nze);
  t_max = 1;
  operation (const float grid d1(-1..nx+2, -1..ny+2, -1..nz+2),
             float grid u1(-1..nx+2, -1..ny+2, -1..nz+2),
             const float grid xx(-1..nx+2, -1..ny+2, -1..nz+2),
             const float grid xy(-1..nx+2, -1..ny+2, -1..nz+2),
             const float grid xz(-1..nx+2, -1..ny+2, -1..nz+2),
             float param dth) {
    float c1 = 9./8.;
    float c2 = -1./24.;
    float d = 0.25 * d1[x,y,z] + d1[x,y-1,z] + d1[x,y,z-1] + d1[x,y-1,z-1]);
    u1[x,y,z; t+1] = u1[x,y,z; t] + (dth / d) * (c1 * (xx[x, y,z]-xx[x-1,y,z]+xy[x,y, z]-xy[x,y-1,z]+xz[x,y,z ]-xz[x,y,z-1]) +
                                          c2 * (xx[x+1,y,z]-xx[x-2,y,z]+xy[x,y+1,z]-xy[x,y-2,z]+xz[x,y,z+1]-xz[x,y,z-2])
  }
}
```

Listing 2: The stencil specification for uxx1.

The actual stencil computation is defined within the operation. The arguments to the operation are the input and output grids needed for the computation; an additional \texttt{const} specifier declares a grid to be constant in time, i.e., as not being written to within the operation. Again, the names of the grid identifiers match the ones of the original Fortran code; thus the correct pointers will be passed automatically to the generated implementations. Optionally, the grid size can be specified in round brackets to match the size of the array as it was actually allocated. If this is not done, Parus assumes as small as possible grid sizes, which in this case would lead to incorrect results.

The body of the operation contains the localized, point-wise stencil expression; stencil sweeps, i.e., the spatial iterations, are not programmed explicitly. Listing 2 demonstrates that the stencil specification language allows multiple stencil expressions, which can be assigned to temporary variables, which can be used in later expressions. Parus will also recognize \texttt{c1} and \texttt{c2} as constants and substitute the actual values into the stencil expressions.

In addition to the parameters related to the domain size and the pointers to the grids, the generated stencil kernel function expects the Strategy parameters to be passed; i.e., in the cache blocking Strategy shown in Listing 1 the function expects that the cache block sizes are passed as arguments. The idea is to tune these parameters in an offline tuning phase. Thus they are known at compile time of the application and will be passed to the kernel functions as constants. The offline tuning will be described in more detail in Section 6.
5. Experimental Testbed

Before moving on to auto-tuning the Strategy parameters for best kernel performance and to the application performance benchmarks, this chapter gives a brief description of the target hardware, namely a Cray XK6 system installed at the Swiss National Supercomputing Center. Each of the 176 hybrid compute nodes is equipped with one 8-module AMD Opteron 6272 “Interlagos” processor and one NVIDIA Tesla X2090 GPU. The GPUs were not used in this study and hence will not be discussed here.

The compute nodes are interconnected with Cray’s Gemini interconnect, which directly connects to the Hyper-Transport interconnect rather than the PCIe bus as in commodity interconnects, and thus can deliver superior node-to-node bandwidth. The interconnect is arranged in a 3D torus topology.

The Interlagos CPUs implement AMD’s recent “Bulldozer” microarchitecture, which has been completely re-designed. The chips are manufactured in 32 nm silicon on insulator high-κ metal gate process technology. Each Interlagos socket contains two dies, each of which contains four so-called “modules.” Conceptionally, a module is a cross-over of traditional full cores and hardware multithreading as seen in Intel’s Hyper-Threading Technology. A module contains two full separate out-of-order dedicated integer cores, but only one floating point unit. Furthermore, the early pipeline stages (instruction fetch and decode) and the L2 cache are shared among the cores to minimize silicon area and for power efficiency reasons. Each integer core has its own L1 cache. AMD’s Turbo Core Technology allows cores to run up to 500 MHz faster if all cores are utilized (or even more if not) than their base clock frequency of 2.1 GHz, depending on the detected workload [14].

The floating point unit features two 128-bit processing engines, which can be combined into one 256-bit engine to support the AVX instruction set [15]. In addition, the processing engines are capable of doing fused multiply-adds, and thus the processor supports AMD’s FMA4 instruction set.

Interlagos’s memory hierarchy is composed of a 16 KB four-way L1 data cache per integer core, a 64 KB two-way L1 instruction cache per module, a 2 MB 16-way per-module unified L2 cache, and 8 MB of L3 cache, which is shared among the four modules on a die. As each die has its own memory controller, one Interlagos socket is inherently a NUMA architecture. The available bandwidth to DRAM was measured with the STREAM benchmark [16]; around 27 GB/s using both memory controllers were observed.

6. Kernel-Level Tuning

In order to determine the optimal values for the per-kernel tuning parameters, in a first step all the kernels were auto-tuned in an offline tuning phase. The resulting parameter values were substituted into the final application code. All the performance measurements shown in this paper are based on codes compiled with GCC 4.6.2 compilers (gcc, gfortran) with the -O3 optimization flag. Only single precision performance numbers are shown as AWP-ODC does all the calculations in that precision mode. The double precision kernel performance is slower by the usual factor of 2.

Since we optimize the application for the AMD Interlagos architecture, which is a NUMA architecture, NUMA-aware data initialization is vital when multiple threads are run on one node. The machine implements the “first touch” memory allocation policy, i.e., a set of data elements will reside in the DRAM of the die on which the thread runs that first writes it. Thus, the data used for computation by a particular thread should also be initialized by that thread. As we choose the cache blocking Strategy (cf. Listing 1), the auto-tuning parameters influence how the data is distributed to the threads, and therefore influence how the data is initialized. All kernels operate on the same data; hence, we need to find a parameter set which minimizes the execution time of all kernels simultaneously. To this end, in each auto-tuner iteration, all the kernels are benchmarked, and the weighted sum of execution times is minimized. As weights we chose the percentages of total execution time of the respective kernel, as shown in Table 1. Although the absolute percentage values depend on the local domain size, their ratios remain constant.

Fig. 2 shows a per-kernel performance comparison between the simultaneously (subfigure (a)) and individually (subfigure (b)) tuned kernels. The major horizontal axis shows a representative of each set of kernels, the minor horizontal axis shows the number of threads. The dots in the upper part of subfigure (a), whose values have to be read on the right vertical axis, show the performance degradation of the simultaneously tuned versions with respect to the individually tuned ones. The degradation remains below 5% for the family of the uxx1 kernel, except for the
The scaling behavior across OpenMP threads is summarized in Table 2. It shows the absolute performance numbers and comparison between different versions of Parus-generated, auto-tuned C codes as well as manually OpenMP-threaded Fortran versions. (a) Performances when all the kernels are tuned simultaneously, i.e., when trying to find one common parameter set for all kernels and performance degradation with respect to the performance of individually tuned kernels. The black dots show the performance of the original Fortran code, which was parallelized with OpenMP. (b) Performances of individually auto-tuned kernels. The green dots show the performance of the manually optimized and auto-tuned version of the Fortran code. The black markers show the performance limit due to memory bandwidth and the kernels’ arithmetic intensities.

Table 2: SSE performance in GFlop/s (first number) and thread scaling behavior. The speedup with respect to the performance of one thread and the speedup with respect to the original manually OpenMP-threaded Fortran code is printed in brackets.

<table>
<thead>
<tr>
<th>Kernel</th>
<th>1 Thread</th>
<th>2 Threads</th>
<th>4 Threads</th>
<th>8 Threads</th>
<th>16 Threads</th>
</tr>
</thead>
<tbody>
<tr>
<td>uxx1</td>
<td>2.25 (1.00; 2.12)</td>
<td>3.74 (1.66; 2.01)</td>
<td>6.56 (2.92; 1.81)</td>
<td>9.89 (4.40; 1.51)</td>
<td>19.86 (8.83; 2.00)</td>
</tr>
<tr>
<td>xy1</td>
<td>2.66 (1.00; 1.73)</td>
<td>3.88 (1.46; 1.60)</td>
<td>6.24 (2.34; 1.35)</td>
<td>8.44 (3.17; 1.05)</td>
<td>16.19 (6.08; 1.95)</td>
</tr>
<tr>
<td>xyz1</td>
<td>3.26 (1.00; 4.74)</td>
<td>4.54 (1.39; 3.49)</td>
<td>9.03 (2.77; 3.54)</td>
<td>15.60 (4.79; 3.26)</td>
<td>31.53 (9.68; 3.37)</td>
</tr>
</tbody>
</table>

The scaling behavior across OpenMP threads is summarized in Table 2. It shows the absolute performance num-

single-threaded performance, in which case the performance drops by 10%. Similarly, for the xy1 kernel family, the decline remains below 7%, and in case of the xyz1 kernel, there is virtually no noticeable performance loss.

Parus is able to generate vectorized implementation using SIMD (SSE, AVX) intrinsics rather than relying on the compiler to automatically vectorize the code. We generated three versions of the kernel codes: one using SSE intrinsics operating on 128-bit SIMD vectors, and two versions using AVX intrinsics operating on 256-bit SIMD vectors, one with and the other without using the AMD-specific fused multiply-add instruction of the FMA4 instruction set.

Fig. 2 shows the performance differences. The blue bars show the performance in GFlop/s of the C code which was not vectorized using SIMD intrinsics (i.e., we relied on the compiler to do the vectorization), the purple bars show the performance of the SSE version, and the red and yellow bars the AVX versions. Furthermore, the black dots in Fig. 2 (a) show the performance of the original Fortran code, which was parallelized with OpenMP for this experiment: an OpenMP sentinel was inserted before the outer-most stencil kernel loop. The green dots in Fig. 2 (b) show the performance of a manually optimized and auto-tuned version of the kernel written in Fortran. The tuned Fortran version features a better parallelization scheme (the one used by Parus) and cache blocking. The results suggest that the Fortran compiler was able to vectorize the smaller uxx1 and xy1 kernels well, but not the larger xyz1 kernel. The black markers show the maximum performance which can be reached due to the available memory bandwidth and the arithmetic intensities of the kernels. Most noticeably in case of the xyz1 kernel, explicit vectorization results in a large performance gain. There is not much performance difference between the SSE and the AVX implementations. This is due to the hardware, as the Interlagos architecture contains two 128-bit SSE engines, which are combined into one 256-bit engine if run in AVX-256 mode. In general, the SSE implementation runs fastest, except when only one thread is used. In this case, AVX+FMA4 outperforms the other implementations by a small margin.

The scaling behavior across OpenMP threads is summarized in Table 2. It shows the absolute performance num-
We use weak scaling for the MPI processes, i.e., we assign equally sized local domains to each MPI process. For the OpenMP scaling, we show both weak and strong scaling. In the former case we fix the workload per thread to 256 × 256 × 128 grid points and the workload per 16-thread compute node to 512^3 grid points by running 16/t MPI processes per node, where t ∈ {1, 2, 4, 8, 16} is the number of OpenMP threads. In the latter case, we fix the local per-node domain size to 256^3 for any number of threads and run one MPI process per node.

We compare two versions of the code: the original code, which has been threaded manually by adding OpenMP sentinels to the outer-most loops of the routines we focused on in this paper, and the code with the original routines substituted by their Parus-generated equivalents. Obviously, as we only parallelized 7 stencil kernels, which, in our benchmarks, cover 61% to 88% of the total execution time in the non-threaded version, we cannot expect linear scaling. We employ a simple performance model to predict the scaling behavior. Assuming that the compute time for the sequential baseline \( T(1) \) is given, for the weak scaling case we approximate the compute time \( T(t) \) with \( t \) threads employed by

\[
T(t) = T(1) \left( \sum p_i + t \left( 1 - \sum p_i \right) \right),
\]

where \( p_i \) is the percentage of compute time spent in kernel \( i \) in the sequential case. This is essentially Amdahl’s Law. Since only these kernels are parallelized, we expect that the rest of the code will take \( t \) times as long to execute as in the sequential case because domain size is scaled by a factor of \( t \). The model is optimistic in the sense that it does not take into account the slowdowns with respect to linear scaling across OpenMP threads and pessimistic in the sense that the sequential workload is assumed to be constant albeit the surface-to-volume ratio changes.

In Fig. 3, on the major horizontal axis the number of threads cooperating on one grid block is varied, on the minor horizontal axis the number of nodes is varied. The vertical axis shows the time in seconds required to compute one time step. Subfigure (a) shows the weak scaling case, subfigure (b) the strong scaling case, respectively. The blue bars show the time required by the auto-tuned Parus-generated code using SSE intrinsics, and the purple bars show the additional time required if the original code is used (which actually includes cache blocking, but was not tuned to the hardware platform). Thus, e.g., in the sequential case, the original code computes a 512^3-sized local domain in just under 2 seconds, whereas the code with the Parus-generated replacements is 47% faster on average and computes one time step in 1.27 seconds. The percentage numbers in the figures show by what amount the execution speed was increased. In the sequential case, AVX+FMA4 outperforms SSE by a small margin. The results obtained from the AVX+FMA4 version of the code are included as yellow dots in Fig. 3.

The dark blue and dark purple lines indicate the timings predicted by Eq. (2). In the weak scaling case, the observed timings meet the expectations quite nicely. The figure suggests that if more than 4 threads are used cooperatively, the performance gain due to Parus is negated by the relatively higher sequential overhead. However, in the strong scaling case, the performance prediction seems to be too pessimistic, although the Parus version virtually fails to scale, possibly due to the small domain sizes. Yet, in the sequential case it yields a 2.3× speedup over the original code.
8. Related Work

Recently, other frameworks specifically crafted for stencil computations have emerged, although their focus is slightly different from the one of Patus. In particular, Panorama [17] was a research compiler for tiling iterative stencil computations in order to minimize cache misses. The Berkeley stencil auto-tuner [18] seeks to substitute an annotated stencil computation in Fortran95 automatically by an optimized version. The Pochoir stencil compiler [19] applies the cache oblivious ideas initially formulated by Frigo and Strumpen [20] to stencil codes with ideally many time steps. Mint [21] targets NVIDIA GPUs as hardware platforms and translates traditional, but annotated, C code to CUDA C and applies hardware-specific optimizations specifically tailored for stencil computations.

More general approaches, which are not limited only to stencil computations, consider tiling of perfectly and imperfectly nested loops in the polyhedral model [22]. Loop transformation and (automatic parallelizing) compiler infrastructures in the polyhedral model include CHiLL [23] and PLuTo [24].

9. Conclusions

In this paper, we presented an approach to automatically generate and optimize stencil kernels extracted from a real-world application, the anelastic wave propagation code AWP-ODC, and to re-integrate them into the full application. We showed that, for the individual stencil kernels, our framework Patus is able to generate scalable threaded implementations leveraging non-trivial parallelization and code optimization strategies — including cache blocking, explicit vectorization, and loop unrolling, — and the auto-tuning methodology.

On the kernel level, we observed that the kernel versions generated by Patus are on average 2.4 times faster than the original, manually optimized and OpenMP-threaded Fortran code; in one particular case the generated version achieves a 4.7× speedup. On the full-application level, we showed that the scaling behavior, when only the main computational kernels are threaded, is as can be expected from Amdahl’s Law. In the weak thread scaling scenario, which keeps the workload per thread constant, the time to compute one time step of the simulation was reduced by 47% on average with respect to the original Fortran code, which was threaded manually for this experiment. However, as only a part of the code was threaded, the time to compute a time step increases in accordance with Amdahl’s Law. Thus, unless larger portions of the code are threaded as well, the performance benefit from threading is limited. In the strong scaling scenario, we observed a speedup of 2.3× due to Patus in the non-threaded setting.

However, in consideration of current hardware trends, threading will become increasingly important. The number of cores per processor will continue to increase (for an exa-scale machine, which the industry strives to produce by the end of the decade, it was projected that it would have to contain thousands of cores per processor), while the amount of memory increases much more slowly. Hence, the memory available to each core will drop, which eventually
necessitates cooperative threading in order to avoid the replication of overhead, which, e.g., is associated with each MPI process.

We hope that Parus can help to automatically introduce threading stencil-based codes, while providing an implementation optimized for the target platform at the same time. Parus is open source software and licensed under the GNU Lesser General Public License. The software can be obtained from http://code.google.com/p/parus/.

Acknowledgments

The authors would like to thank Luis Dalguer (Swiss Seismological Service, ETH Zurich) and Yifeng Cui (High Performance GeoComputing Lab, San Diego Supercomputer Center) for corresponding with the authors and making the AWP-ODC code available. This project was funded within the Petaquake project of the High-Performance and High-Productivity Computing platform initiated by the Swiss National Supercomputing Center.

References


