Comput Sci Res Dev (2011) 26: 205–210
DOI 10.1007/s00450-011-0160-6

SPECIAL ISSUE PAPER

Automatic code generation and tuning for stencil kernels on modern shared memory architectures

Matthias Christen · Olaf Schenk · Helmar Burkhart

Published online: 6 April 2011

Abstract In this paper, we present PATUS, a code generation and auto-tuning framework for stencil computations targeted at multi- and manycore processors, such as multicore CPUs and graphics processing units. PATUS, which stands for "Parallel Autotuned Stencils," generates a compute kernel from a specification of the stencil operation and a strategy which describes the parallelization and optimization to be applied, and leverages the autotuning methodology to optimize strategy-specific parameters for the given hardware architecture.

Keywords Stencil computations · Code generation · Autotuning · High performance computing

1 Introduction

In many numerical codes, ranging from simple PDE solvers to complex AMR and multigrid solvers, the class of stencil computations is a constituent class of kernels. Oftentimes, stencil computations comprise a dominant part of the compute time. Therefore, in order to minimize the time to solution, it is crucial that the stencil kernels make use of the available computing resources as efficiently as possible. However, microarchitectures have become more and more complex and diverse, and, as a consequence, meticulous architecture- and application-specific tuning is required to elicit the machine's full compute power. This not only requires deeper understanding of the architecture, but is also both a time consuming and error-prone process.

M. Christen (✉) · O. Schenk · H. Burkhart
Department of Mathematics and Computer Science, University of Basel, Klingelbergstrasse 50, 4056 Basel, Switzerland
e-mail: m.christen@unibas.ch

The PATUS framework is a code generation and autotuning tool for the class of stencil computations. The idea behind the PATUS framework is twofold: on the one hand it provides a software infrastructure for generating architecture-specific stencil code from a specification of the stencil incorporating domain-specific knowledge that enables optimizing the code beyond the abilities of current compilers, and on the other hand it aims at being an experimentation toolbox for parallelization and optimization strategies. Using a small domain specific language (DSL), the user can define the stencil kernel using a C-like syntax, and can choose from predefined strategies how the kernel is optimized and parallelized, or design a custom strategy to experiment with other algorithms or find a better mapping to the hardware in use. Besides supporting almost arbitrary types of stencils on structured grids and generating code from strategy templates, another goal of PATUS is to be able to support future hardware microarchitectures and programming paradigms. Currently we support traditional CPU architectures using OpenMP for parallelization and NVIDIA CUDA-capable GPUs.

This work is closely related to the more generally applicable approach of loop tiling [4, 5, 7, 9, 10]. Cache-oblivious blocking schemes for iterative stencil computations determining the optimal tile sizes at runtime are proposed in [3, 11]. The autotuning methodology has been applied successfully in diverse libraries and frameworks for various types of kernels which occur frequently in scientific computing, including ATLAS, FLAME, OSKI, FFTW, SPIRAL, and recently in a framework for stencil computations [6]. The customizable strategies are a key feature that sets PATUS apart from other frameworks such as the aforementioned one.

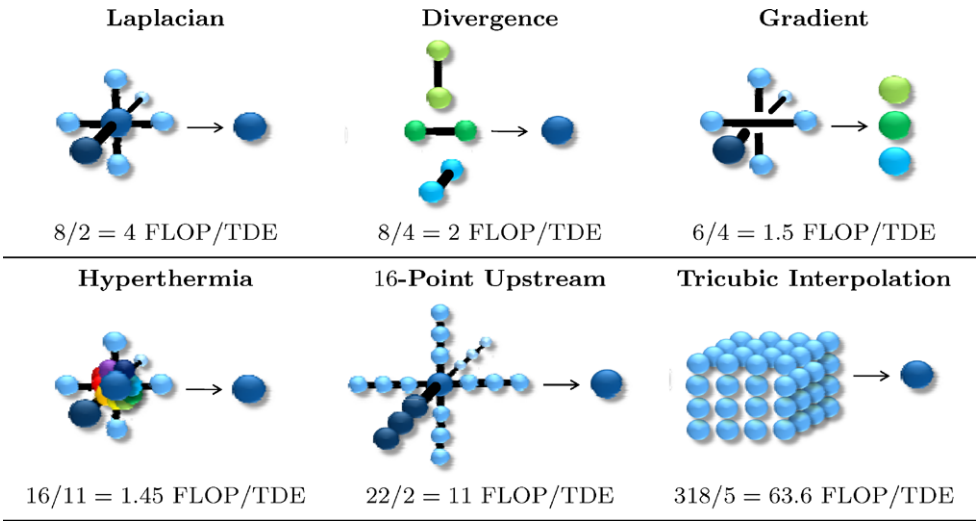

Fig. 1 Stencil examples studied in this paper. The *images* show the structure of the input and output nodes. The numbers are the arithmetic intensities in number of floating point operations (FLOPs) per transferred data element (TDE). The numerator is the actual number of FLOPs per stencil computation, the denominator the number of actually transferred data elements

2 Stencil examples and bandwidth saving algorithms

A stencil is a fixed geometric arrangement defined on a structured grid. A stencil computation assigns the center node a value depending on the values previously assigned to the neighboring nodes in the fixed arrangement. This is done for all the inner nodes of the grid. Examples of applications of stencil computations include finite difference-type PDE solvers and image processing filters.

2.1 Stencil examples

In this paper we concentrate on the stencil examples shown in Fig. 1. These stencil kernels will be used as benchmark examples in Sect. 4. The figure shows a number of examples of stencil structures ("input" nodes to the left of the arrows in the images, and "output" nodes to their right) and highlights their variety. The "Laplacian", "Divergence", and "Gradient" stencils are finite difference discretizations of the corresponding basic differential operator, whereas "Hyperthermia", "Upstream", and "Tricubic Interpolation" come from real world applications; the former comes from a simulation of the temperature distribution within the human body during hyperthermia cancer treatment [1], and the latter two occur as typical examples in the weather forecast code COSMO.

In stencil computations, the number of floating point operations (FLOPs) per grid point is constant. The number of FLOPs typically is low compared to the (constant) number of memory references. I.e., stencil computations have a constant arithmetic intensity with respect to the problem size, unlike, e.g., BLAS3 operations. The arithmetic intensities in FLOPs per transferred data element for the examples is given in the figure. Here, data transfers mean transfers between RAM and caches (CPUs) or global memory and registers (GPUs). Because of the low FLOP rate, we typically expect the performance of stencil computations to be bounded by the available memory bandwidth. Hence, the key for enhancing performance lies in minimizing data transfers.

2.2 Saving bandwidth

There are several approaches to get rid of non-compulsory data transfers and thereby reducing the memory traffic. The key is to reuse data that have been loaded previously, i.e., to exploit data locality.

On cache-based architectures, cache blocking is a well known technique to improve temporal data locality: by decomposing the grid into cache size dependent small subgrids it is ensured that data loaded into the cache are reused before being evicted due to capacity misses.

Iterative stencil computations can benefit from blocking not only in space, but also in time, especially if there is a local memory that can be controlled explicitly by the programmer such as on a GPU. Temporal blocking has the advantage of greater temporal data locality and reduced synchronization overhead. The basic idea is to compute multiple timesteps with all the data kept in local memory and therefore to avoid writing the data back to main memory after one timestep and reloading it again for the next as well as to avoid synchronization within a time block. The technique was previously adapted to different types of hardware architectures and was described in [1, 2, 8].

Another temporal blocking scheme is a "wavefront" parallelization proposed by Wellein et al. [12]. The idea of the wavefront parallelization is having a team of threads cooperate on a chunk of data. While thread i is sweeping through the subgrid, thread $i + 1$ takes the output of thread i to perform its sweep. Hence, each thread in the team calculates one timestep on the same subgrid. Because of the data dependencies, thread $i + 1$ has to wait for thread i to complete the computation of the input data that is needed for the computation before it can start its sweep, making it look like ripples of waves passing through the subgrid.

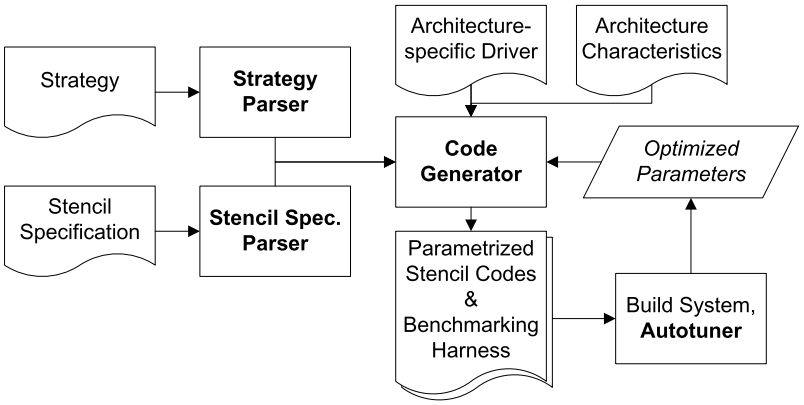

Fig. 2 High-level overview of the software architecture of PATUS. The strategy and the stencil specification are input files, which drive the code generation. The code generator creates a set of parametrized hardware-specific kernels that are executed by the autotuner, which determines the optimal parameter set

3 The PATUS framework

PATUS expects 3 input files to generate a stencil kernel code: The major input, from the user's point of view, being the specification of the stencil operation. E.g., the discrete Laplacian

$$u'_{ijk} = \alpha u_{ijk} + \beta(u_{i-1,j,k} + u_{i+1,j,k} + u_{i,j-1,k} + u_{i,j+1,k} + u_{i,j,k-1} + u_{i,j,k+1})$$

is specified in the PATUS stencil DSL like so:

```
stencil laplacian {
  operation (double grid u,
    double param alpha, double param beta) {
    u[x, y, z; t+1] =
      alpha * u[x, y, z; t] +
      beta * (
        u[x-1, y, z; t] + u[x+1, y, z; t] +
        u[x, y-1, z; t] + u[x, y+1, z; t] +
        u[x, y, z-1; t] + u[x, y, z+1; t]);
  }
}
```

The second input is a "strategy," which describes how the kernel source is actually generated: It describes parallelization methods or a bandwidth saving algorithm by means of a second DSL. The description is independent both of the stencil and of the hardware architecture and the programming model used. Strategies are also the interfaces to the autotuner: strategy parameters (e.g., blocking sizes) are picked up by the autotuner, which tries to find the values for which the code has the best performance.

PATUS provides predefined strategies, but the user can develop own strategies and thereby experiment with other parallelization and optimization approaches. The DSL is expressive enough so that the afore-mentioned bandwidth saving algorithms can be implemented.

The third input file describes various aspects of the hardware for which the code is generated, e.g., it specifies the programming model, thereby selecting the code generator back-end, whether or not the hardware requires explicit data transfers to local stores, whether explicit SIMDization is required, etc.

3.1 The software infrastructure

PATUS is built from four core components: the parsers for the two input files, the stencil definition and the strategy; the code generator, which is driven by the third input file, the architecture specification; and the autotuner. Figure 2 gives a high-level overview over the software architecture.

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 used in the kernel routine. The code generator transforms the strategy AST to C code and "instantiates" the stencil, i.e., replaces the formal grids and stencil calls in the strategy by the actual identifiers and stencil computation. Moreover, it handles data transfers to local memory if required and performs optimizations such as explicit SIMDization and loop unrolling. Back-ends for shared memory CPU systems using OpenMP for parallelization and CUDA-capable single-GPUs systems have been implemented so far.

In order for the autotuner to perform the benchmarks, the code generator also creates a benchmark harness. The problem size and the autotuning parameters are expected as command line arguments by the benchmark harness. After building the executable from the kernel code and the benchmark harness, the autotuner seeks to find the optimal configuration for the parameters by repeatedly running the program with the autotuning parameters varying according to some search method. The PATUS autotuner supports a variety of search methods.

4 Experimental performance results

The performance benchmark were carried out on an Intel Nehalem (Intel Xeon E7540 "Beckton") architecture, the

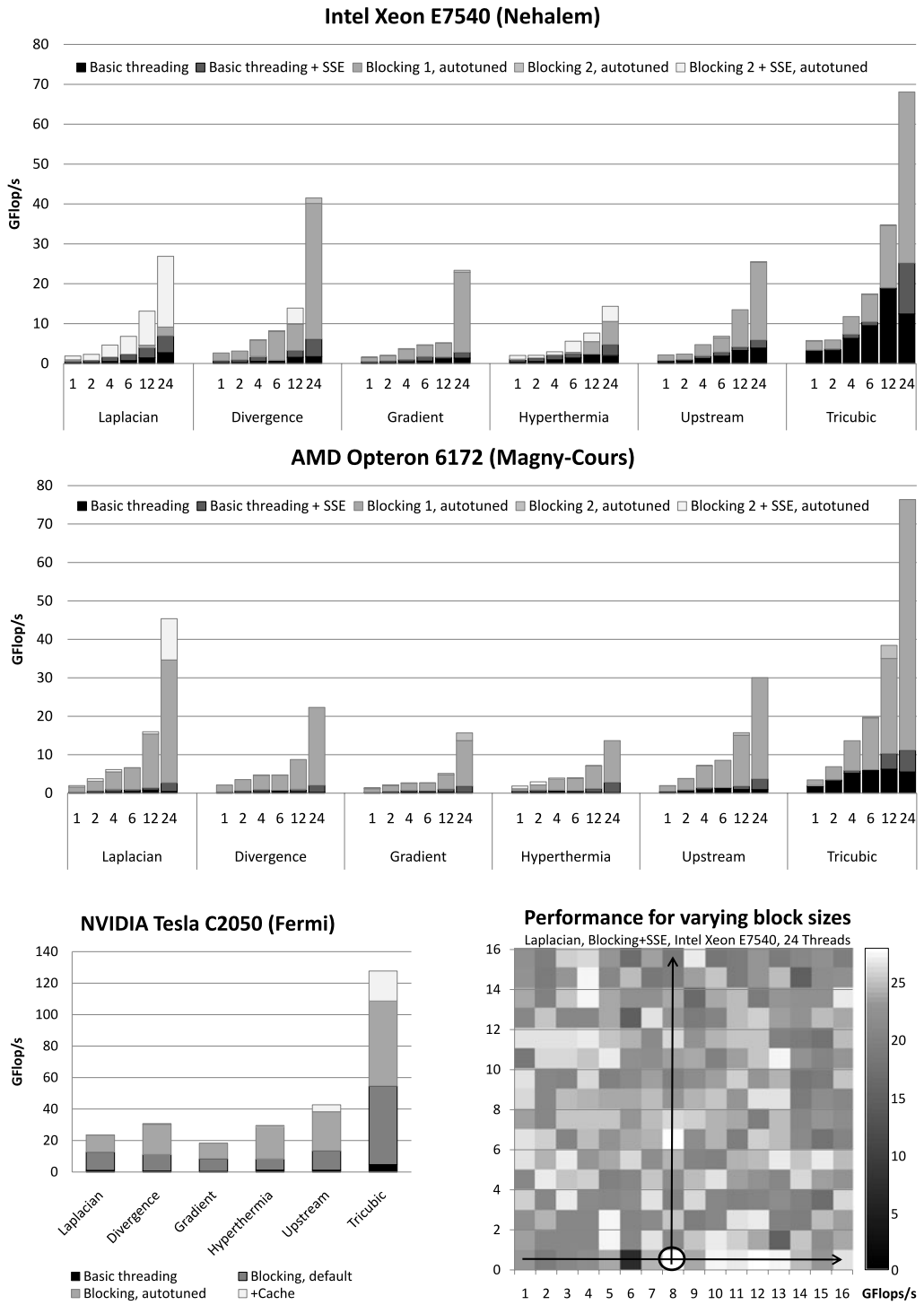

Fig. 3 Performance results for 6 types of stencils on 1 to 24 threads of the Intel Nehalem and the AMD Magny-Cours architectures, and on the NVIDIA C2050 Fermi GPU for naïve threading and blocking strategies. The low arithmetic intensity stencil kernels—Laplacian, Divergence, Gradient, and Hyperthermia—were calculated using single precision floating point numbers; the kernels with high arithmetic intensity—Upstream and Tricubic—were calculated in double precision. The figure to the *lower right* shows the performance variations for block sizes varying in two dimensions. The autotuner picks the block yielding the best performance by searching along one axis, fixing the size with the best performance and continuing the search along the next axis as symbolized by the *arrows* (Powell search method). The size with the best performance has been *highlighted* in the figure

Table 1 Characteristics of the hardware architecture used in the performance benchmarks

	Intel Xeon E7540	AMD Opteron 6172	NVIDIA Tesla C2050
Cores	2×6	4×6	14
Concurrency	24 HW threads	24 HW threads	448 ALUs
Clock	2 GHz	2.1 GHz	1.15 GHz
L1 Data Cache	32 KB	64 KB	$48 + 16$ KB
L2 Cache	256 KB	512 KB	–
Shared L3 Cache	18 MB	6 MB	–
Avg. Shared L3/HW Thread	1.5 MB	1 MB	–
Measured Bandwidth (STREAM)	35.0 GB/s	53.1 GB/s	79.3 GB/s

AMD Magny-Cours (AMD Opteron 6172), and an NVIDIA Tesla C2050 GPU Computing Processor. Some architecture characteristics are summarized in Table 1.

We performed performance benchmarks using the autotuned code with a set of strategies applied to the stencil kernels from Sect. 2.1. In all the plots, the performance numbers of first four stencils (Laplacian, Divergence, Gradient, and Hyperthermia) are single precision GFLOP/s; Upstream and Tricubic are double precision GFLOP/s. We used a 128^3 sized grid for all the stencil examples. Five runs with one timestep each were performed and timed. The reported performance numbers are average numbers.

The "basic threading" strategy only parallelizes the stencil computation without doing any blocking. On the CPUs two different blocking strategies differing in the numbers of blocking levels were applied. Both display similar performance results after autotuning on both architectures. Generating explicit SSE intrinsics (instead of relying on the compiler to do the vectorization) proved to be beneficial for the 7-point stencils (Laplacian and Hyperthermia). The slightly higher absolute performance numbers on the Magny-Cours is due to its superior bandwidth for the bandwidth-limited cases and slightly higher clock rate for the compute-bound case ("Tricubic").

On the GPU, besides using a parallelization using the same basic strategy as for the CPUs, a blocked strategy with two parallelism levels was chosen. The graph shows substantial speed improvements when the thread block sizes are chosen carefully (in our case by means of the autotuner) over default 4^3 thread block sizes. The bar labeled "+Cache" shows the performance improvement from increasing the GPU cache size from 16 KB to 48 KB per streaming multiprocessor. The GPU code generation is still work in progress, and the figure is merely included to highlight the fact that PATUS is able to generate CUDA code, while code optimizations still need to be improved.

Figure 3 to the lower right, finally, shows the performance for block sizes varying in two dimensions. It is the autotuner's job to picks the block yielding the best performance. In Fig. 3, the Powell search method is shown, which searches along one axis, fixing the size with the best performance and continuing the search along the next axis.

5 Conclusion

We presented PATUS, a code generation and autotuning framework for general stencil computations. It is thought of as both a productivity tool and a tool for experimenting with parallelization and optimization strategies. We have shown that the approach works for both modern multi- and many-core architectures, and the performance numbers demonstrate the potential of leveraging non-trivial strategies and the autotuning methodology. The current framework still has limitations that we intend to overcome in the future. The framework will be publicly available under an open source type license.

References

1. Christen M, Schenk O, Neufeld E, Paulides M, Burkhart H (2010) Manycore stencil computations in hyperthermia applications. In: Scientific computing with multicore and accelerators. CRC Press, Boca Raton, pp 255–277
2. Datta K, Kamil S, Williams S, Oliker L, Shalf J, Yelick K (2008, to appear) Optimization and performance modeling of stencil computations on modern microprocessors, SIAM Rev
3. Frigo M, Strumpen V (2005) Cache oblivious stencil computations. In: ICS'05: proceedings of the 19th annual international conference on supercomputing. ACM, New York, pp 361–366
4. Goumas G, Athanasaki M, Koziris N (2003) An efficient code generation technique for tiled iteration spaces. IEEE Trans Parallel Distrib Syst 14:1021–1034
5. Hall M, Chame J, Chen C, Shin J, Rudy G, Khan M (2010) Loop transformation recipes for code generation and auto-tuning. In: Gao G, Pollock L, Cavazos J, Li X (eds) Languages and compilers for parallel computing. Lecture Notes in Computer Science, vol 5898. Springer, Berlin, pp 50–64
6. Kamil S, Chan C, Oliker L, Shalf J, Williams S (2010) An autotuning framework for parallel multicore stencil computations. In: IEEE International Parallel & Distributed Processing Symposium (IPDPS), pp 1–12
7. Li Z, Song Y (2004) Automatic tiling of iterative stencil loops. ACM Trans Program Lang Syst 26(6):975–1028

8. Meng J, Skadron K (2011) A performance study for iterative stencil loops on GPUs with ghost zone optimizations. Int J Parallel Program 39:115–142. doi:10.1007/s10766-010-0142-5
9. Renganarayanan L, Kim D, Rajopadhye S, Strout M (2007) Parameterized tiled loops for free. ACM SIGPLAN Not 42:405–414
10. Rivera G, Tseng C (2000) Tiling optimizations for 3D scientific computations. In: Supercomputing, ACM/IEEE 2000 conference
11. Strzodka R, Shaheen M, Pajak D, Seidel H (2010) Cache oblivious parallelograms in iterative stencil computations. In: ICS'10: proceedings of the 24th ACM international conference on supercomputing, pp 49–59
12. Wellein G, Hager G, Zeiser T, Wittmann M, Fehske H (2009) Efficient temporal blocking for stencil computations by multicore-aware wavefront parallelization. In: COMPSAC(1), pp 579–586

Matthias Christen received his MS degree in Mathematics from the University of Basel, Switzerland, in 2006. Currently he is a PhD student in Computer Science at the University of Basel. His research interests are in code generation, autotuning, and high-performance computing.

Olaf Schenk received a diploma in mathematics from the Karlsruher Institute of Technology, Germany, a PdD degree from the Swiss Federal Institute of Technology (ETH) and Venia Legendi (Habilitation) from the University of Basel, Switzerland. The research of Olaf Schenk concerns algorithmic and architectural problems in the field of computational mathematics, scientific computing and high-performance computing. In these areas, he has published more than 60 peer-reviewed journal articles and conference contributions. He is an IEEE Senior Member, and a SIAM Member. He received a highly-competitive IBM Faculty Award on Cell Processors for Biomedical Hyperthermia Applications in 2008 and was the finalist of the International Itanium Award in 2009 in the area of computational intensive applications.

Helmar Burkhart is a Computer Science Professor at the University of Basel since 1987. He received a diploma in Computer Science from the University of Stuttgart, Germany, and a PdD degree and Venia Legendi (Habilitation) from the Swiss Federal Institute of Technology (ETH) Zurich, Switzerland. He have held several positions such as President of the Swiss Informatics Society SI/Swiss Chapter of the ACM (1990–1992), member of the expert group Swiss Priority Programme in Informatics Research (1991–1996), and cofounder and board member of the SPEEDUP association. His research interests include parallel and distributed processing, web technologies, and e-learning.