2011 IEEE International Parallel & Distributed Processing Symposium

PATUS: A Code Generation and Autotuning Framework For Parallel Iterative Stencil Computations on Modern Microarchitectures

Matthias Christen, Olaf Schenk, Helmar Burkhart
Department of Mathematics and Computer Science
University of Basel, Switzerland
{ m.christen | olaf.schenk | helmar.burkhart } @unibas.ch

***Abstract*—Stencil calculations comprise an important class of kernels in many scientific computing applications ranging from simple PDE solvers to constituent kernels in multigrid methods as well as image processing applications. In such types of solvers, stencil kernels are often the dominant part of the computation, and an efficient parallel implementation of the kernel is therefore crucial in order to reduce the time to solution. However, in the current complex hardware microarchitectures, meticulous architecture-specific tuning is required to elicit the machine's full compute power. We present a code generation and auto-tuning framework PATUS for stencil computations targeted at multi- and manycore processors, such as multicore CPUs and graphics processing units, which makes it possible to generate compute kernels from a specification of the stencil operation and a parallelization and optimization strategy, and leverages the autotuning methodology to optimize strategy-dependent parameters for the given hardware architecture.**

***Keywords*-stencil computations; code generation; autotuning; high performance computing**

I. INTRODUCTION

Stencil calculations comprise an important class of kernels in many scientific computing applications ranging from simple PDE solvers to constituent kernels in multigrid methods as well as image processing applications. Often, in such types of solvers, the major part of the computation time is spent in a stencil kernel. Therefore it is important in order to minimize the time to solution, that the stencil kernels make use of the available computing resources as efficiently as possible. However, in the current complex hardware microarchitectures, meticulous architecture-specific tuning is required to elicit the machine's full power. This not only requires deeper understanding of the architecture, but is also both a time consuming and error-prone process.

Libraries and code generators for other important kernels in scientific computing, including dense and sparse linear algebra and discrete transforms, successfully adapt autotuning as a means to automatically select the code that delivers the best performance from a family of codes based on automatic performance benchmarks.

The PATUS framework is a code generation and autotuning tool for the class of stencil computations. PATUS stands for "Parallel AutoTUned Stencils". It is the result of generalizing the insights gained from performance studies of a kernel from a real-world application involving different kinds of architectures.

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 permits to optimize 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 small domain specific languages, 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 in order to experiment with other algorithms or find a better mapping to the hardware in use. This is one of the key features in which PATUS differs from other code generation and autotuning frameworks for stencil codes, such as the one proposed by Kamil [1].

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. The modular code generator back-end allows adding support for new hardware by defining hardware-specific characteristics and implementing code generator methods for a few communication and synchronization primitives.

Currently we support traditional CPU architectures using OpenMP for parallelization and NVIDIA CUDA-capable GPUs.

II. RELATED WORK

Autotuning has been applied successfully in diverse libraries and frameworks for various types of kernels which occur frequently in scientific computing, including ATLAS [2] and FLAME [3] for dense linear algebra, OSKI [4] for sparse linear algebra, FFTW [5] and SPIRAL [6] for signal processing transforms, and recently in a framework for stencil computations [1].

The search space is either a (possibly parameterized) code base from which the autotuner in an offline process determines the version and parameters that display the best performance on a given architecture (ATLAS, FFTW using

1530-2075/11 $26.00 © 2011 IEEE
DOI 10.1109/IPDPS.2011.70

a priori knowledge, Kamil's stencil autotuning framework), by describing the search space algebraically, i.e., by having a symbolic representation of the algorithms and applying transformations under which the meaning of an algorithm is invariant (SPIRAL, FLAME), or by statistical sampling and tuning at runtime (OSKI).

A different approach to automatic generation of efficient kernels is based on hardware models and deriving models for performance. Loop tiling approaches [7], [8], [9], [10], [11] and compilers using the polyhedral model fall into this category. Loop tiling is a more general approach that does not only apply to stencil loops; also, tiles are not restricted to be rectangular as in our work, but can be parallelotopes. Finding parallelotope sizes is a nontrivial task that involves solving systems of linear inequalities. [7], [10] propose efficient algorithms within this context to this end. The polyhedral model is used for both determining good tile sizes as well for auto-parallelization [12].

Time blocking schemes for stencil computations and corresponding performance evaluations on a variety of hardware platforms are discussed in [13], [14]; [15] implements the method on GPUs, and also gives an elaborate performance model for the method on that architecture. Another time blocking method is presented in [16] and evaluated on shared memory CPU architectures. Frigo and Strumpen [17] as well as Strzodka et al. [18] propose cache-oblivious blocking schemes for iterative stencil computations, which determine the optimal tile sizes at runtime.

III. Stencil Computations and Bandwidth Saving Algorithms in Patus

A stencil is a geometric structure defined on a structured grid. It consists of a particular arrangement of nodes around a center node. In a stencil computation, the values of this fixed arrangement of nodes are used to update the value of the center node. The computation is carried out for each node in the grid. Stencil computations have been recognized as one of the fundamental compute patterns and are also called the "structured grid motif" [19]. They arise for instance in finite difference-type PDE solvers, they also occur in multigrid methods: smoothing, restriction, and prolongation operators are essentially stencil computations; also in image processing filters such as blur or edge detection are stencil computations.

A. Stencil Examples

In this section we give a few examples of stencils, for which we also will carry out the performance benchmarks presented in section VI-B. We take 3 stencils that arise from discretizations of basic differential operators and 3 stencils that come from discretizations of PDEs that are used to model and solve real-world problems.

1) The 3D discrete Laplacian,

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

arises from the finite difference discretization of the Laplace operator $\Delta := \frac{\partial^2}{\partial x_1^2} + \frac{\partial^2}{\partial x_2^2} + \frac{\partial^2}{\partial x_3^2}$, $\Delta : C^p(\Omega) \longrightarrow C^{p-2}(\Omega)$, for an open set $\Omega \subseteq \mathbb{R}^3$.

2) The 3D discrete divergence operator,

$$\begin{aligned} u_{ijk} &= \alpha\,(X_{i+1,j,k} - X_{i-1,j,k}) + \\ &\quad \beta\,(Y_{i,j+1,k} - Y_{i,j-1,k}) + \\ &\quad \gamma\,(Z_{i,j,k+1} - Z_{i,j,k-1}) \end{aligned}$$

discretizes the divergence operator $\nabla\cdot := \frac{\partial}{\partial x_1} + \frac{\partial}{\partial x_2} + \frac{\partial}{\partial x_3}$ that maps a differentiable vector field to a function.

3) The 3D discrete gradient operator,

$$\begin{bmatrix} X_{ijk} \\ Y_{ijk} \\ Z_{ijk} \end{bmatrix} = \begin{bmatrix} \alpha\,(u_{i+1,j,k} - u_{i-1,j,k}) \\ \beta\,(u_{i,j+1,k} - u_{i,j-1,k}) \\ \gamma\,(u_{i,j,k+1} - u_{i,j,k-1}) \end{bmatrix},$$

is the finite difference discretization of the gradient operator $\nabla := \left(\frac{\partial}{\partial x_1}, \frac{\partial}{\partial x_2}, \frac{\partial}{\partial x_3}\right)^T$ that maps differentiable functions to a vector field.

4) The stencil we used for simulating the temperature distribution within the human body during hyperthermia cancer treatment [20], [14],

$$\begin{aligned} u'_{i,j,k} &= u_{i,j,k}\,(a_{i,j,k}u_{i,j,k} + b_{i,j,k}) + c_{i,j,k} \\ &\quad + d_{i,j,k}u_{i-1,j,k} + e_{i,j,k}u_{i+1,j,k} \\ &\quad + f_{i,j,k}u_{i,j-1,k} + g_{i,j,k}u_{i,j+1,k} \\ &\quad + h_{i,j,k}u_{i,j,k-1} + l_{i,j,k}u_{i,j,k+1} \end{aligned}$$

stems from a finite volume discretization of the parabolic Pennes bioheat equation [21],

$$\rho C_{\mathrm{p}} \frac{\partial u}{\partial t} = \nabla\cdot(k\nabla u) - \rho_{\mathrm{b}} W(u) C_{\mathrm{b}} (u - T_{\mathrm{b}}) + \rho Q + \tfrac{\sigma}{2}\|\mathbf{E}\|^2,$$

5) Finally, we look at two types of stencils occurring in the weather forecast code COSMO [22]: an asymmetric 16-point stencil in 3D with three nodes parallel to each of the axes in negative direction and two nodes in positive direction. It is found in an upstream scheme common in weather codes.

6) The last example is a 3D tricubic interpolation: a $4 \times 4 \times 4$-point stencil,

$$u'_{ijk} = \sum_{\eta,\iota,\kappa} \left(w_\eta(a_{ijk}) w_\iota(b_{ijk}) w_\kappa(c_{ijk}) u_{i+\eta,j+\iota,k+\kappa}\right)$$

for $-1 \leq \eta, \iota, \kappa \leq 2$, given weight functions w_ℓ, and coefficient grids a, b, c.

B. Arithmetic Intensity

In stencil computations, the number of floating point operations per grid point is constant. The number of FLOPs typically is low compared to the number of memory references, which is also constant. I.e., stencil computations have a constant arithmetic intensity with respect to the problem size, unlike, e.g., BLAS3 operations. Because of the low FLOP rate, we typically expect the performance of stencil computations to be limited by the available memory bandwidth. Hence, the key for maximum performance lies in minimizing data transfers. The traversal of the grid can be designed in such a way that memory transfers are reduced with respect to the naïve traversal (i.e., a D-fold nested loop over the entire grid, D being the dimensionality of the domain). A few examples are given in section III-D.

The upper bounds for the arithmetic intensities of the stencils under consideration are:

Kernel	FLOPs	Data Elts	Arith. Intensity (FLOPs/data element)
Laplacian	8	2	4.00
Divergence	8	4	2.00
Gradient	6	4	1.50
Hyperthermia	16	11	1.45
16-point Upstream	22	2	11.00
Tricubic	318	5	63.60

The upper bounds are calculated from the compulsory data transfers: we assume that by bringing one grid point into local memory the neighboring grid points are also brought automatically into memory. This follows the hardware data loading pattern, which is in blocks than rather by individual elements, e.g., cache lines on CPUs and the typical shared memory loading pattern on CUDA-programmed GPUs.

C. Specifying a Stencil in PATUS

The previous examples show the diversity of stencil operators. The discrete Laplacian is an example for a stencil with same input and output grid structures. We are particularly interested in such types of stencils because they can be applied iteratively (a technique that is commonly used to solve elliptic PDEs iteratively or to solve time-dependent PDEs as in example 4). This property allows us to apply an optimization called "temporal blocking." Other stencil operators such as the divergence or the gradient operators have vector-valued input or output structures, respectively.

Another stencil characteristic is the structure of the stencil. Examples 1 and 4 are 7-point stencils in 3D. Examples 5 and 6 are higher-order stencils with considerably more points: the tricubic interpolation is a 64-point stencil in three dimensions.

Other stencil computation characteristics are boundary treatment (typically imposed by boundary conditions of the PDE) and grid traversal (e.g. Jacobi, Gauss-Seidel, colored iterations).

PATUS provides a means to specify these characteristics by a domain-specific language. The following is the specification for the 3D discrete Laplacian:

```
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]);
  }
}
```

For the boundary treatment, the stopping criteria, and the grid traversal we allot **`boundary`**, **`stoppingcriteria`**, and **`filter`** within the **`stencil`** specification. However, for the time being we restrict ourselves to boundary treatment that corresponds to Dirichlet boundary conditions, a constant number of iterations (if the stencil can be iterated), and Jacobi iterations. Support for other types will be added in the future.

In PATUS, we assume that each grid used in the computation is a contiguous data array, i.e., that grids are given as a structure of arrays, as opposed to the "array of structures" format, in which each grid point would be encapsulated in a structure with members corresponding to the individual grids. Hence, the stencil specification for the hyperthermia stencil (example 4) looks like so:

```
stencil hyperthermia
{
  operation (float grid u, float const grid c[9])
  {
    u[x, y, z; t+1] = ...
  }
}
```

`u` corresponds to the input and output grids u and `c` to the 9 coefficients $a, \ldots, l$ that vary in space (hence they are specified as **`grid`**s), but are constant in time, which is indicated by the **`const`** keyword.

D. Strategies

In the PATUS framework, a "strategy" describes a parallelization methods or a bandwidth saving algorithm by means of a second domain specific language. The description is independent of the stencil and also of the hardware architecture and the programming model used.

We assume that the order in which the stencil computations on the individual grid points are executed does not change the result of the entire computation. Therefore, any permutations of the elements of the loop index space is legal. In particular, any reordering transformation is legal. All the blocking techniques described below as well as parallelization make use of this fact. The order in which the

statements within the original loops are executed (say, the sequential, naïve version consisting of a D-fold loop nest), is *not* preserved.

On cache-based architectures, cache blocking is a well known technique to improve temporal data locality: by decomposing the domain into cache size dependent small subdomains it is ensured that data loaded into the cache is reused before it is evicted due to capacity misses. Furthermore, since the domain is already decomposed, the computation is easily parallelized since the computed value on one grid point does not depend on the computed values on other grid points of the output grid. Cache blocking is expressed with the following strategy code:

```
strategy cacheblock (grid u, auto dim cb)
{
  // iterate over time steps
  for t = 1 .. stencil.t_max
  {
    // iterate over subdomain
    for subgrid v(cb) in u[:; t] parallel
    {
      for plane pln in v[:; t]
        for point pt in pln[:; t]
          v[pt; t+1] = stencil (v[pt; t]);
    }
  }
}
```

The strategy parameters are the formal grid `u`, which will be replaced by all the grids required for the stencil computation by the code generator, and an **`auto`** parameter `cb`, the cache block size. Strategies are both templates for the code generator and interfaces to the autotuner. Adding an **`auto`** specifier to a parameter means that it will be picked up by the autotuner, which then tries to find the optimal value for that parameter giving the best performance. The generated kernel function code will have the cache block parameter `cb` in its signature. The code generator exposes it as a command line argument in the benchmark executable.

In this strategy, we iterate over all the timesteps required by the computation. **`stencil`**`.t_max` is simply a placeholder for the number of iterations given explicitly or implicitly (via some stopping criterion) in the stencil specification. In the `v` loop, the grid `u` (at timestep `t`) is broken into subgrids of size `cb`, and, by virtue of the **`parallel`** keyword, each thread is assigned subgrids `v` over which it iterates. The formal **`stencil`** call is replaced by the actual stencil expressions, taking into account the actual grids required for the computation and their local sizes for correct indexing.

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.

For illustration, consider a regular 7-point stencil in 3D. Typically, data is loaded into the local memory in planes orthogonal to the slowest iteration direction. To compute one result plane, three planes are required. Once the data for the first local timestep resides in the local memory, the first local timestep is computed. The result plane is stored in the local memory, and as soon as three result planes have been calculated, they can be used as input for the next timestep. Note that the size of the planes shrink after each timestep due to the data dependencies. Hence, in favor of easier parallelization, we load and calculate overlapping halo regions redundantly. The drawback, however is, that the added overhead can only be compensated if the local memory is large enough.

This "circular queue" technique is most effective when we can control data movement from and to the main memory explicitly, i.e., on an architecture with a software-managed cache such as the Cell BE, on which speed improvements of about $2\times$ have been shown for specific stencils [23], [20], [14]. The circular queue temporal blocking can be formulated as the following PATUS strategy code:

```
strategy circularqueue (grid u,
  auto int timeblocksize, auto dim cb)
{
  // do all the time steps
  for t = 1 .. stencil.t_max by timeblocksize
  {
    // perform a domain decomposition and
    // process the subdomains in parallel
    for subgrid v(cb) in u[:; t] parallel
    {
      // process the subdomains planewise
      for plane p in v[:; t]
      {
        // perform local timesteps on each plane
        for t0 = t .. t + timeblocksize - 1
        {
          // apply the stencil
          for point q in p[:; t]
            p[q; t0+1] = stencil (p[q; t0]);
        }
      }
    }
  }
}
```

Other temporal blocking schemes include the wavefront parallelization proposed by Wellein et al. in [16], and cache-oblivious schemes [17], [18].

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. Specifically, thread 0 reads the input array and writes its result to a temporary array that is consumed by thread 1. Thread 2 reads from the temporary array and writes to another temporary array, etc. The last thread reads from its temporary input array and writes the result to the output array. Currently, PATUS does not have support for temporary data yet. This, however, will be added in the future, which makes it possible to implement strategies like the wavefront parallelization.

PATUS only supports strategies with static assignment of data to threads, i.e., dynamic assignments found for instance in cache oblivious schemes [17], [18] are not supported.

Although strategies per se are designed to be architecture independent, the code generated from a particular strategy might not perform equally well on each of the hardware architectures under consideration. In fact, depending on the architecture characteristics, a strategy might degenerate or even not work at all. For instance, the wavefront parallelization is designed for a multicore CPU architecture and makes use of the fact that cores have shared caches. On a platform like the Cell BE, where compute entities cannot access "foreign" local stores, the strategy will only work in the degenerate case with one thread per team, a case in which the benefits of the algorithm are obviously nullified.

If not done by the user, it is the task of PATUS's autotuner to single out strategies that do not perform well on the architecture under consideration.

IV. Prerequisites: Mapping to the Hardware

A. Hardware Model

Inspired by the OpenCL execution model [24], we take a hierarchic view on the execution units. We allow them to be indexed by multidimensional indices, similar to OpenCL's "NDRange" index spaces. This guarantees that there is an optimal mapping to architectures that have hardware support for multidimensional indexing or have multidimensional indices built into the programming model such as CUDA or OpenCL. We call a level in the hierarchy a "parallelism level". The dimension of the indices are allowed to differ in each parallelism level.

Optionally, each parallelism level can have a local memory that is only visible to the execution units within and below that parallelism level, provided that the execution units on the levels below do not have their own local memory. The data transfers to the local memories can be either implicit or explicit, i.e., managed by hardware or software. Furthermore, we permit both synchronous and asynchronous data transfers.

We view the execution units as SIMD entities that support all the usual arithmetic operations and optionally support fused multiply-adds via intrinsics.

According to this model, a shared-memory CPU architecture has one parallelism level with local memory (cache) with implicit data transfer, and a CUDA-capable GPU has two parallelism levels, streaming multiprocessors and streaming processors – or thread blocks and threads, respectively. The thread block level has an explicit transfer local memory, namely the per-multiprocessor shared on-chip memory. Nowaday's CPUs typically support SIMD via the streaming SIMD extensions (SSE), while, from this perspective, the GPU's streaming processors are scalar units.

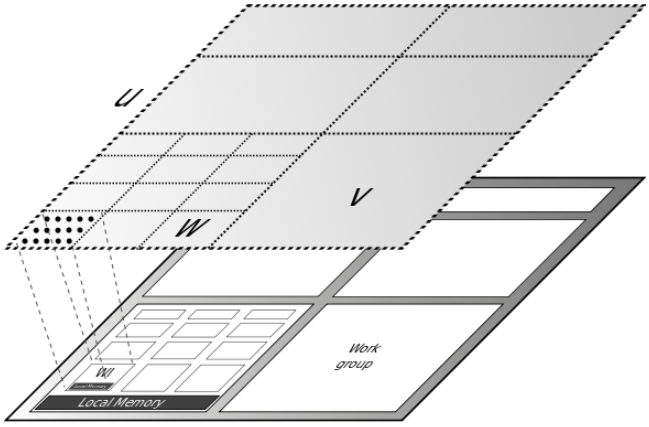

Figure 1. Mapping between data and hardware. Both hardware architecture (bottom layer) and data (top layer) are viewed hierarchically: the domain u is subdivided into v and w, the hardware architecture groups parallel execution units on multiple levels together.

B. Mapping Strategies

Domain decomposition and mapping to the hardware is implicitly given by a strategy. Every subgrid iterator, e.g.,

```
for v(blk) in u[:;t]
  ...
```

decomposes the domain subgrid into subgrids of smaller size `blk`. When in addition the **`parallel`** keyword is used on a subgrid iterator, the loop is assigned to the next parallelism level (if there is one available), and each of the iterator boxes is assigned to an execution unit on that parallelism level. All the loops within the iterator also belong to the same parallelism level (i.e., are executed by the units on that level), until another parallel loop is encountered in the loop nest.

If a parallelism level requests explicit data copies, memory objects are allocated for an iterator box as "late" as possible: since local memories tend to be small, the iterator within the parallelism level with the smallest boxes, i.e., as deep down in the loop nest as possible (such that the loop still belongs to the parallelism level), is selected to be associated with the memory objects. The sizes of the memory objects are derived from the box size of that iterator, and data from the memory objects associated with the parallelism level above are transferred. In the case that the iterator contains a stencil call within a nested loop on the same parallelism level, the iterator immediately above a point iterator ("**`for point`**

`p` **`in`** `v[:; t] ...`") is selected, if there is one, or if there is no such iterator, the iterator containing the stencil call is selected.

The strategy excerpt

```
for subgrid v(blk1) in u[:;t] parallel
  for subgrid w(blk2) in v[:,t] parallel
    for point p in w[:,t]
      ...
```

and the resulting data decomposition and the ideal hardware mapping are visualized in Fig. 1. The upper layer shows the hierarchic domain decomposition of u into v and w. The bottom layer shows an abstraction of the hardware architecture with 2 parallelism levels, work groups and work items, which both can have a local memory. By making the v loop in the strategy **`parallel`**, it is assigned to the first parallelism level, labeled "work groups" in the figure (following the OpenCL convention). And by making the nested w loop **`parallel`**, this, in turn, is assigned to the second parallelism level, the work items within a work group. The points p in w are all assigned to the same work item that "owns" w.

C. Mapping Dimensions

The dimensionality of the data is implied by the specification of the stencil. Let D denote the data dimensionality. To compute an index into the D-dimensional data, we use a unified approach that "extends" the hardware indices of dimension d_i (i being the parallelism level) to intermediate indices of dimension $\min\{D, \max_i\{d_i\}\}$ by emulating possibly missing dimensions using the last coordinate, then creating a global index combining all the parallelism levels, and finally, in the same way, extending the intermediate index to the full D-dimensional index.

V. The Patus 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, and the autotuner. Patus is written in Java and uses Coco/R [25] as a parser generator, and the Cetus framework ([26], [27]) provides Java classes for the abstract syntax trees (AST) for the strategies and for the generated code. The computer algebra system Maxima [28] is interfaced for symbolic computations and as a powerful expression simplifier. Fig. 2 gives a high-level overview over the software architecture.

The parsers read the input files and produce internal representations for the stencil definition and the strategy. The strategy is transformed to an abstract syntax tree that is used as a template by the code generator. The stencil specification is turned to the graph representation of the stencils. These structures are passed as input to the code generator, along with an additional configuration that describes the characteristics of the hardware and the programming model used to program the architecture and specifies the code generation

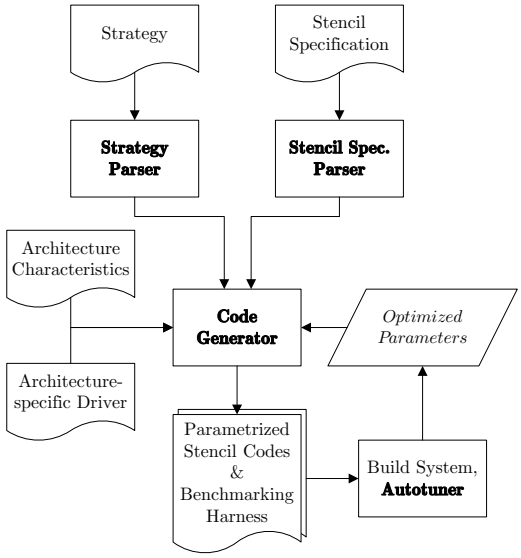

Figure 2. High-level overview of the architecture of Patus.

back-end. 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.

In order for the autotuner to perform the benchmarks, the code generator also creates a benchmark harness from a programming model-specific "templated" implementation of a driver source file that allocates the grids required for the stencil computation, initializes them, and invokes and measures the runtime of the stencil kernel. The problem size and the autotuning parameters are expected as command line arguments by the driver. 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.

A. Current Limitations

Currently, we impose limitations on the grid traversal and on the boundary conditions. We consider only Jacobi iterations and restrict ourselves to Dirichlet-type boundary conditions at the boundaries of the rectilinear domain, i.e., we apply the stencil to each point in the interior of the domain leaving the values on the boundary points fixed, and assume that the order in which the stencils are applied is irrelevant to the result. Irregular domains and non-uniform grids can be emulated by introducing additional grids that encode the domain shape and non-uniformity of the grid. In fact, this is done in the hyperthermia stencil example. Furthermore, we only consider shared memory architectures at this time, specifically multicore CPU and single-GPU systems.

B. The Code Generator

In this section we describe the tasks carried out by the code generator. It transforms the strategy AST to C code and "instantiates" the stencil, i.e., replaces the strategy stencil function placeholder by the real stencil computation and replaces all the references to the formal grid by the grids that are actually required to carry out the computation. Moreover, it handles data transfers to local memory if required and performs optimizations such as explicit SIMDization and loop nest unrolling.

1) Parallelization: From the strategy, the code generator first resolves the **`parallel`** keyword and extracts a new abstract syntax tree from the original strategy AST for the code that each thread executes. By the nature of our problem, we only deal with data parallelism; each thread executes the same code, but a on different set of data.

Specifically, the **`parallel`** keyword is removed from a subgrid iterator and a subgrid index is computed from the thread ID. There are two modes: the "multicore mode" assumes that each thread is assigned multiple subgrids on which the thread will work. The "manycore mode" in turn assumes that there are enough threads to assign each thread at most one of the subgrids.

2) Stencil Representation and Memory Objects: A stencil is represented as a triple $(\Sigma_i, \Sigma_o, \Phi)$, where Σ_i is a finite set of input stencil nodes, Σ_o is the set of output stencil nodes, and Φ is an expression that formally assigns each output stencil node an arithmetic expression in the input stencil nodes. A stencil node is an index $(x_1, \ldots, x_n; t; j) \in \mathbb{Z}^n \times \mathbb{Z} \times \mathbb{N}$ with spatial component $(x_1, \ldots, x_n)$, a temporal index t, and a counting index i.

The stencil representation describes the data dependencies between input and output nodes. Together with a strategy subgrid iterator, it provides a tool for determining how memory objects have to be created, for calculating indices into these memory objects, and for analyzing the reuse of memory objects. The index calculation engine needs the sizes of the memory objects, which are defined by subgrid iterators and the widths of halo layers, which are computed from the stencil representation.

In order to determine the shape of memory objects and how they are reused, we use the following mechanism. A subgrid iterator with an iterator box of size $(s_1, \ldots, s_D)$ induces projections $\pi, \rho, \overline{\rho} : \mathbb{Z}^D \twoheadrightarrow \mathbb{Z}^D$, which we write as diagonal matrices in $\mathbb{Z}^{D \times D}$, defined by

$$\pi_{ii} := \begin{cases} 1 & \text{if } s_i = 1, \\ 0 & \text{otherwise,} \end{cases}$$

ρ_{ii} contains exactly one 1 at the place where π has the first non-zero entry, and $\overline{\rho}$ is the diagonal matrix with $\overline{\rho}_{ii} = 1 - \rho_{ii}$. The matrices act on the spatial component of a stencil node in the obvious way. An equivalence class of stencil nodes under the projection π defines a memory object. ρ gives the "reuse direction", i.e., a vector parallel to an axis along which memory objects can be reused in space. Finally, projection under $\overline{\rho}$ groups memory objects into reuse classes, each of which need to be treated independently.

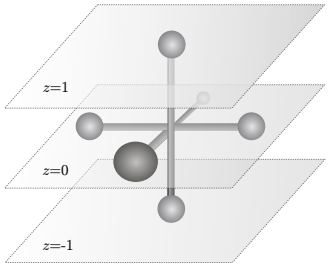

Figure 3. Stencil nodes of a 3D 7-point stencil contained in memory objects (planes $z = -1, 0, 1$) induced by a plane iterator.

For instance, consider the plane iterator "**`for plane`** p ..." in 3D, which has an iterator box of size $(b_x, b_y, 1)$ for some width $b_x > 1$ and length $b_y > 1$. (Plane iterators in PATUS are always oriented such that the indices within the plane move faster than the plane index, and by convention, the x-axis is the unit stride direction.) This iterator proceeds plane-wise for $z = 0, 1, 2, \ldots$. It induces the projection $\pi = \begin{pmatrix} 0 & 0 & 1 \end{pmatrix}$, and stencil nodes whose spatial components differ only in the x and y coordinates, but have equal z coordinates, fall into the same equivalence class, hence the same memory object. Here, $\rho = \pi$, so z is our reuse direction: when proceeding from plane $z = 0$ to $z = 1$, the planes $z = 0$ and $z = 1$ can be reused while only one new plane, $z = 2$ needs to be loaded. Applying $\overline{\rho} \circ \pi$ to the set of stencil nodes projects them all to the trivial space $\{0\}$, i.e., there is only one reuse class.

3) Transferring data: Data is transferred from a parent memory object to a child when the child is associated with a new parallelism level. Since overlapping computation and communication is important in bandwidth limited kernels, PATUS creates code for asynchronous data transfers through a pipelined preloading scheme – if the hardware supports asynchronous communication – by preloading the front memory objects in the reuse direction.

4) Instantiating a stencil: All the formal grid variables and the formal stencil call are replaced by the memory objects corresponding to grids and grid iterator variables on the local memory of the particular parallelism level and by the actual stencil expressions, respectively.

5) SIMDization: PATUS can explicitly SIMDize the stencil computation. The domain specific knowledge can help to create more efficient SIMD code than a vectorizing compiler can produce. In fact, we observed that the SIMD code generated by PATUS outperformed the code that was automatically vectorized by the Intel compiler. (We used version 11.1 of the Intel compiler to compile both generated scalar code and code generated with SSE intrinsics.) Also, the programming

environment might require manual SIMDization: e.g., the compiler for the SPU code on the Cell BE platform does not vectorize code automatically, but due to the architecture design, SIMD is essential.

Since PATUS's memory objects are SIMD-aware, indexing a memory object within the code generator will calculate the correct index into the array of SIMD vectors.

The penalty for using SIMD intrinsics are alignment restrictions. Firstly, SIMD vector must be correctly aligned in memory (at multiples of the size of a SIMD vector), and secondly, element-wise arithmetic operations also need to respect that alignment; e.g., it is not possible to directly add the two vectors (v_0, v_1, v_2, v_3) and (v_1, v_2, v_3, v_4) because the second vector is misaligned. Note that stencil computations typically require exactly such kinds of operations. The way around this is to combine two (correctly aligned) SIMD vectors, (v_0, v_1, v_2, v_3) and (v_4, v_5, v_6, v_7), and use shuffling intrinsics to extract the desired shifted vector (v_1, v_2, v_3, v_4). We want to avoid too many shuffling operations, because they can become a potential bottleneck.

For SIMD vector lengths $\lambda > 1$ the memory objects are aligned in such a way that data elements corresponding to interior grid points are aligned at an address divisible by $\lambda \cdot \text{size(type)}$. The size of the inner memory object cells in unit stride direction (and only in unit stride direction, since SIMD only affects the unit stride direction) is padded such that the number of cells is a multiple of λ. Also the sizes of the halos at the left and right of the memory object are padded to become multiples of λ. By this we ensure to minimize shifting data elements for SIMD operations during the calculation: shifting is only required when accessing neighbors in unit stride direction whose offset from the center node is not divisible by λ, but not when accessing neighbors in non-unit stride directions.

PATUS's SIMD code generator is independent of the particular SIMD implementation in hardware (e.g. SSE, Altivec, AVX, etc.). The SIMD vector length and the intrinsics for the basic operations and functions can be defined for each base datatype in the architecture description, which then replace the arithmetic operators and mathematical functions in the stencil expressions. Alternatively, the corresponding methods in the concrete back-end code generation implementations can be overwritten.

6) Loop nest unrolling: This optimization explicitly unrolls a loop *nest* in one go, rather than individual loops at a time. For instance, unrolling each loop within a double nested loop by 2 would read as:

```
for i = i_0 .. i_1                          for i = i_0 .. i_1 by 2
  for j = j_0 .. j_1          ⇝               for j = j_0 .. j_1 by 2
    S(i,j)                                      S(i,j)
                                                S(i,j+1)
                                                S(i+1,j)
                                                S(i+1,j+1)
```

Note that the order in which the statements S are executed is not preserved by this transformation.

Loop nest unrolling gives rise to a fine-grained blocking of the code that can be beneficial on the register level because of register reuse and possibly increased instruction level parallelism. For this reason, this type of blocking is also referred to as register blocking [13]. We apply this type of blocking only to strategy loops in the innermost level which contain one and only one call to the strategy **`stencil`** function placeholder.

Loop nest unrolling cannot be achieved by parameterizing the code since it changes the code structure itself. Therefore, the loop nest unrolling optimization is not exposed to the strategy interface. As an internal structural code generation optimization, each loop nest unrolling configuration gives rise to a new "code branch" that will be implemented in its own kernel function. In the main kernel function it is decided which branch is taken based on the loop unrolling parameter.

7) Data initialization: On NUMA CPU architectures, most of the time careful data initialization is key to good performance for bandwidth limited computations. On such systems, typically the "first touch" page placement policy is implemented, i.e., the socket on which a thread first touches (e.g., writes to) a page will own that page, meaning that it will be placed in the DRAM close to that socket. Whenever a thread running on another socket accesses the data, that piece of data has to be transferred over an interconnect (Intel's Quick Path Interconnect or AMD's HyperTransport) to the socket that requested it. Using the interconnect is typically significantly slower than the direct access to DRAM.

A NUMA-aware data initialization therefore ensures that memory locations are first touched by the thread that is going to reference these locations later in the computation. (A prerequisite for this to be effective is that threads are not allowed to migrate between sockets. In order to make optimal use of the per-core caches, threads should also be pinned to cores.)

PATUS creates a data initialization routine that is derived from the stencil kernel and writes initial values to each of the grids that will be used in the computation.

8) Back-ends: The code generator back-ends constitute the mechanism how the PATUS framework supports different types of architectures and programming models. It consists of two parts: a configuration file in the XML format that describes certain characteristics of the architecture, and a back-end code generator, which directly generates platform-specific code that cannot be easily described in the configuration file.

Currently, back-ends for shared memory CPU systems using OpenMP for parallelization and CUDA-capable single-GPUs systems have been implemented.

C. Autotuning

The task of the autotuning module is to try to find the configuration with minimal running time for all the **auto** parameters exposed in the strategy definition and for the internal not exposed parameters (loop nest unrolling factors, padding) used by the code generator. Essentially, this is a multi-dimensional integer programming problem, and the fact that we cannot assume any structure of the solution space adds to the difficulty of finding the global minimum. The autotuners in projects such as ATLAS [2], FLAME [3], FFTW [5], SPIRAL [6], or the autotuning framework for stencils developed at the Berkeley National Lab [1] use gradient-free optimization methods such as Powell search (i.e., searching for a minimum along a fixed axis, then fixing the value for which the minimum along that axis was found and proceed to the next axis), exhaustive search of a subspace (after pruning the full space using heuristics), the Nelder-Mead simplex search [29], dynamic programming (FFTW and SPIRAL) or stochastic optimization methods including random search, evolutionary search, or hill climbing. Machine learning techniques have also been suggested [30].

The PATUS autotuner is configurable such that any optimization method can be plugged in. If the search space is sufficiently small, exhaustive search can be used, otherwise we currently can resort to a multi-run Powell search, Nelder-Mead, or evolutionary algorithms.

The autotuner expects the path to the executable and a set of parameter ranges as input. It runs the benchmark executable with a parameter set, reads the resulting timings, and repeats with updated parameters based on the benchmark results and the search strategy. The autotuner can execute the benchmark program both on the local node on which also the autotuner runs or submit a job to a batch system and read the timings from the file system once the job completes.

VI. Experimental Performance Results

A. Testbed Architectures

In this section we give an overview over the hardware platforms that have been used to carry out the performance benchmarks and optimizations.

1) Intel Nehalem: The Intel Xeon E7540 "Beckton" belongs to the family of the recently released Nehalem architectures, which differs drastically from the previous generations in that the frontside bus, which used to connect processors via a north-bridge chip to memory, has been forgone in favor of an on-chip memory controller. This feature effectuates a considerably higher bandwidth compared to older generation systems.

The chip is manufactured in 45 nm technology. The system is a dual-socket hexacore architecture running at 2 GHz clock frequency, and we used the CPUs in hyperthreading mode, so there are 24 hardware threading contexts available. Each of the cores is equipped with a 32 KB L1 data cache and a shared 256 KB L2 cache. The six cores on one socket share an 18 MB L3 cache.

2) AMD Magny-Cours: AMD's Magny-Cours processor is a 12 core architecture manufactured in 45 nm SOI technology, that in fact consists of two hexacore dies within one package. The cores are out-of-order superscalar processors that can fetch and decode up to three x86-64 instructions per cycle.

Each core has its own L1 and 512 KB L2 caches, and all the cores on a die share a 6 MB L3 cache. Each of the two dies has two DDR3 memory channels and four HyperTransport 3.0 links. This implies that the package itself is a NUMA architecture: each die is a NUMA node, each having its own memory controller. Two of the per-die HyperTransport links are used to connect the dies internally. Hence, the package exposes four memory channels and four HyperTransport links. The maximum theoretical memory bandwidth per socket is 25.6 GB/s.

AMD's solution to the cache coherence problem in multiprocessor settings in the Magny-Cours architecture is HT Assist also known as the Probe Filter, a cache directory that keeps track of all the cache lines that are in the caches of other processors in the system and therefore eliminates the need to broadcast probes. If activated, the cache directory reserves 1 MB of the L3 cache for its use [31].

3) NVIDIA Fermi: The GPU we used for our benchmarks, a Tesla C2050 GPU Computing Processor, is one of the first versions of NVIDIA's Fermi architectures. It features 448 cores ("streaming processors") that are packaged into 14 multiprocessors with 32 cores each. (From another viewpoint, the C2050 is a 14 core architecture with each core being a 32-way SIMD unit, since all the streaming processors within a scheduling unit carry out the same instruction.) The C2050 runs at a clock rate of 1.15 GHz. The theoretical single precision peak performance therefore is around 1 TFlop/s, and the double precision peak performance is about 500 GFlop/s. Note that in the Fermi architecture, the double precision performance has been greatly improved over the previous GPU generations.

Other new architectural features include a larger local memory, which is now 64 KB per multiprocessor, and ECC protection of the DRAM. The local memory acts both as the "traditional" local memory, which required explicitly programmed data movement, as well as a cache. It can be configured to act as a 16 KB cache and provide 48 KB of explicit local memory or 48 KB of cache and 16 KB of local memory.

The GPU is equipped with 3 GB of on-board GDDR5 memory. The GPU supports CUDA compute capability 2.0, and we used version 3.10 of the CUDA SDK and runtime.

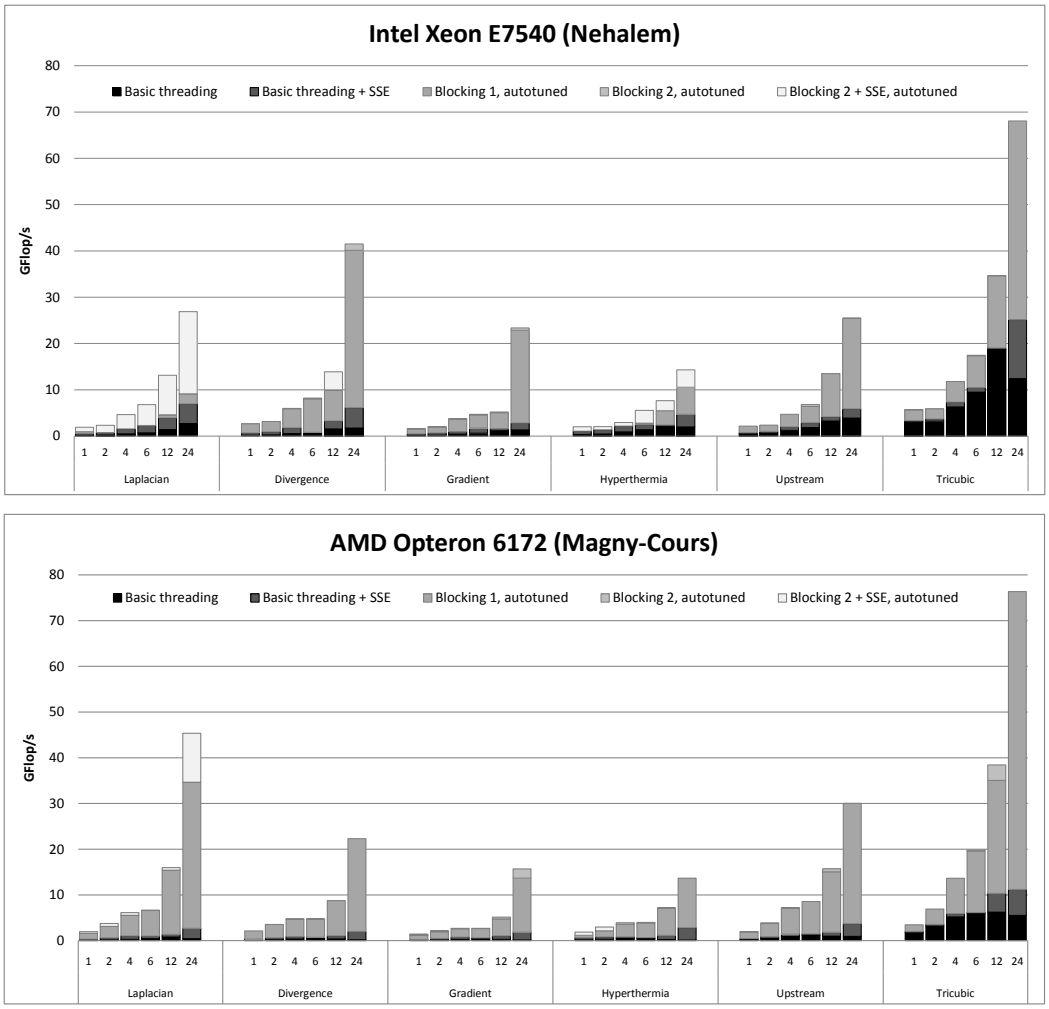

Figure 4. Performance results for 6 types of stencils on 1 to 24 threads of the Intel Nehalem and the AMD Magny-Cours architectures. The Laplacian, Divergence, Gradient, and Hyperthermia stencils were calculated using single precision floating point numbers, Upstream and Tricubic were calculated in double precision.

B. Stencil Performance Benchmarks

We performed performance benchmarks using the auto-tuned code with a set of strategies applied to the stencil kernels discussed in section III-A. We show the results in a common graph per architecture despite the fact that the first four stencils (Laplacian, Divergence, Gradient, and Hyperthermia) were done in single floating point precision, while Upstream and Tricubic used double precision arithmetic.

We used a $128 \times 128 \times 128$ problem for all the stencil examples. Five runs, after an initial warm-up run, with one timestep each were performed and timed. For the benchmarks of the stencils which can be applied iteratively (Laplacian, Hyperthermia, Upstream, Tricubic), this is equivalent to one run with 5 timesteps. The reported performance numbers are average numbers.

The thread affinity was configured to use a "compact" scheme filling cores first, then dies, and then sockets. On the Magny-Cours, this causes to double the available DRAM bandwidth when going from 6 to 12 threads (when going from using one die to two dies in a socket), and again when going from 12 to 24 threads (when going from one to two sockets). On the Nehalem, the DRAM bandwidth is doubled when going from one to two sockets, i.e. from 12 to 24 threads. The graphs show that, except for the Gradient stencil on the Nehalem, this provides enough bandwidth for an almost linear speedup.

Another valid scheme is the "scatter" scheme, which fills the resources in the reverse order, i.e., it places one thread on a socket until all sockets are filled, then adds threads to the free dies on each socket, before filling the cores entirely. Particularly on the Magny-Cours, using the "scatter" scheme, the performance bars would show an

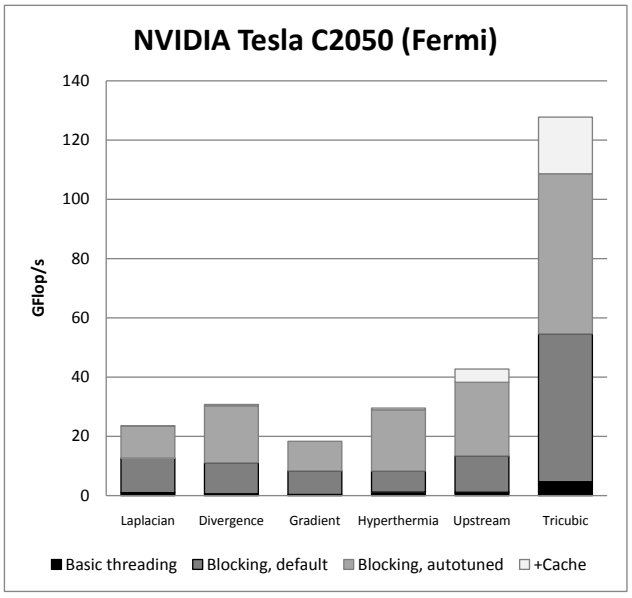

Figure 5. Performance results for 6 stencil types on the NVIDIA C2050 Fermi GPU using full resources. The Laplacian, Divergence, Gradient, and Hyperthermia stencils were calculated using single precision floating point numbers, Upstream and Tricubic were calculated in double precision.

almost linear speedup until the dies are filled (up to 4 threads) and then eventually flatten out.

The "basic threading" strategy only parallelizes the stencil computation without doing any blocking. This means that consecutive grid points are assigned to threads with consecutive thread IDs. For the initialization routine, this implies that – assuming the first touch page placement policy is used – pages are assigned more or less randomly to NUMA nodes, depending on which thread is the fastest to executes the assignment statement, hence the memory affinity is suboptimal. In the computation false sharing is an inhibition to the performance, i.e., threads have cache lines in common to which they write. Explicit SIMDization ("basic threading + SSE") alleviates this problem.

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. For one of the blocking strategies, code with explicit SSE intrinsics was generated (instead of relying on the compiler to do the SIMDization). This proved to be beneficial for the 7-point stencils (Laplacian and Hyperthermia).

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 block sizes (by default, we used $4 \times 4 \times 4$ thread blocks). The bar labeled "+Cache" shows the performance improvement from increasing the GPU cache size from 16 KB to 48 KB per streaming multiprocessor. Data transfers over the PCIe bus were not included in the measurements. The GPU results are still preliminary, and we are striving for optimizations to further exploit the hardware architecture.

VII. Conclusion and Future Work

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, such as bandwidth-saving algorithms: it is for both programmers in need of an efficient implementation of a stencil kernel for a given hardware architecture, but who do not want to care about hardware-specific tuning, and for domain experts who want to experiment. The modular architecture of the system allows to add new components, such as back-ends for other and future hardware. The PATUS framework will soon be publicly available under a liberal open source-style license.

We gave examples for stencil kernels occurring in real-world applications and presented performance numbers for various strategies on recent multicore CPU architectures, an Intel Nehalem CPU and the recently released AMD Magny-Cours, as well as on a NVIDIA Fermi GPU. The performance numbers show the great potential of leveraging an autotuning methodology to find optimal parameters for a given strategy and hardware architecture. We also discussed some implementation details of the code generator.

The current framework still has a few limitations that we intend to overcome in the future. In future work we will suspend the restrictions on the boundary conditions and grid traversal, and more importantly, we will study how the strategy concept can be extended to distributed memory systems.

Acknowledgment

The authors would like to acknowledge the Swiss National Supercomputing Centre (CSCS) for providing access to the compute resources used to carry out the benchmarks.

References

[1] S. Kamil, C. Chan, L. Oliker, J. Shalf, and S. Williams, "An Auto-tuning Framework For Parallel Multicore Stencil Computations," in *IEEE International Parallel & Distributed Processing Symposium (IPDPS)*, April 2010, pp. 1 –12.

[2] R. C. Whaley and J. Dongarra, "Automatically Tuned Linear Algebra Software," in *SuperComputing 1998: High Performance Networking and Computing*, 1998.

[3] R. A. van de Geijn and E. S. Quintana-Ortí, *The Science of Programming Matrix Computations*. `www.lulu.com`, 2008.

[4] R. Vuduc, J. W. Demmel, and K. A. Yelick, "OSKI: A library of automatically tuned sparse matrix kernels," *Journal of Physics: Conference Series*, vol. 16, no. 1, p. 521, 2005.

[5] M. Frigo and S. G. Johnson, "The Design and Implementation of FFTW3," *Proceedings of the IEEE*, vol. 93, no. 2, pp. 216–231, 2005, special issue on "Program Generation, Optimization, and Platform Adaptation".

[6] M. Püschel, J. M. F. Moura, J. Johnson, D. Padua, M. Veloso, B. Singer, J. Xiong, F. Franchetti, A. Gacic, Y. Voronenko, K. Chen, R. W. Johnson, and N. Rizzolo, "SPIRAL: Code generation for DSP transforms," *Proceedings of the IEEE, special issue on "Program Generation, Optimization, and Adaptation"*, vol. 93, no. 2, pp. 232– 275, 2005.

[7] G. Goumas, M. Athanasaki, and N. Koziris, "An Efficient Code Generation Technique for Tiled Iteration Spaces," *IEEE Transactions on Parallel and Distributed Systems*, vol. 14, pp. 1021–1034, 2003.

[8] M. Hall, J. Chame, C. Chen, J. Shin, G. Rudy, and M. Khan, "Loop Transformation Recipes for Code Generation and Auto-Tuning," in *Languages and Compilers for Parallel Computing*, ser. Lecture Notes in Computer Science, G. Gao, L. Pollock, J. Cavazos, and X. Li, Eds., vol. 5898. Springer Berlin / Heidelberg, 2010, pp. 50–64.

[9] Z. Li and Y. Song, "Automatic tiling of iterative stencil loops," *ACM Trans. Program. Lang. Syst.*, vol. 26, no. 6, pp. 975–1028, 2004.

[10] L. Renganarayanan, D. Kim, S. Rajopadhye, and M. M. Strout, "Parameterized Tiled Loops for Free," *SIGPLAN Not.*, vol. 42, pp. 405–414, June 2007. [Online]. Available: http://doi.acm.org/10.1145/1273442.1250780

[11] G. Rivera and C. Tseng, "Tiling optimizations for 3D scientific computations," in *Supercomputing, ACM/IEEE 2000 Conference*, 2000.

[12] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan, "A Practical Automatic Polyhedral Parallelizer and Locality Optimizer," in *Proc. ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation (PLDI 08)*, 2008.

[13] K. Datta, S. Williams, V. Volkov, J. Carter, L. Oliker, J. Shalf, and K. Yelick, *Scientific Computing with Multicore and Accelerators*. CRC Press, 2010, ch. Auto-tuning Stencil Computations on Multicore and Accelerators, pp. 219–253.

[14] M. Christen, O. Schenk, E. Neufeld, M. Paulides, and H. Burkhart, *Scientific Computing with Multicore and Accelerators*. CRC Press, 2010, ch. Manycore Stencil Computations in Hyperthermia Applications, pp. 255–277.

[15] J. Meng and K. Skadron, "A Performance Study for Iterative Stencil Loops on GPUs with Ghost Zone Optimizations," *International Journal of Parallel Programming*, vol. 39, pp. 115–142, 2011, 10.1007/s10766-010-0142-5. [Online]. Available: http://dx.doi.org/10.1007/s10766-010-0142-5

[16] G. Wellein, G. Hager, T. Zeiser, M. Wittmann, and H. Fehske, "Efficient temporal blocking for stencil computations by multicore-aware wavefront parallelization," in *COMPSAC (1)*, 2009, pp. 579–586.

[17] M. Frigo and V. Strumpen, "Cache oblivious stencil computations," in *ICS '05: Proceedings of the 19th annual international conference on Supercomputing*. New York, NY, USA: ACM, 2005, pp. 361–366.

[18] R. Strzodka, M. Shaheen, D. Pajak, and H.-P. Seidel, "Cache oblivious parallelograms in iterative stencil computations," *ICS '10: Proceedings of the 24th ACM International Conference on Supercomputing*, pp. 49–59, 2010.

[19] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick, "The landscape of parallel computing research: a view from Berkeley," Electrical Engineering and Computer Sciences, University of California at Berkeley, Tech. Rep. UCB/EECS-2006-183, December 2006.

[20] M. Christen, O. Schenk, E. Neufeld, P. Messmer, and H. Burkhart, "Parallel Data-Locality Aware Stencil Computations on Modern Micro-Architectures," in *IEEE International Parallel & Distributed Processing Symposium (IPDPS)*, May 2009, pp. 1 –10.

[21] H. H. Pennes, "Analysis of Tissue and Arterial Blood Temperatures in the Resting Human Forearm," *J Appl Physiol*, vol. 1, no. 2, pp. 93–122, 1948.

[22] Core documentation of the COSMO-model. [Online]. Available: http://cosmo-model.cscs.ch/content/model/documentation/core/default.htm

[23] K. Datta, S. Kamil, S. Williams, L. Oliker, J. Shalf, and K. Yelick, "Optimization and Performance Modeling of Stencil Computations on Modern Microprocessors," *SIAM Review*, 2008, to appear.

[24] Khronos OpenCL Working Group, *The OpenCL Specification*, 8 December 2008.

[25] H. Mössenböck, M. Löberbauer, and A. Wöß. The Compiler Generator Coco/R. [Online]. Available: http://www.ssw.uni-linz.ac.at/coco

[26] Cetus – A Source-to-Source Compiler Infrastructure for C Programs. [Online]. Available: http://cetus.ecn.purdue.edu/

[27] H. Bae, L. Bachega, C. Dave, S.-I. Lee, S. Lee, S.-J. Min, R. Eigenmann, and S. Midkiff, "Cetus: A Source-to-Source Compiler Infrastructure for Multicores," in *Proceedings of the 14th Int'l Workshop on Compilers for Parallel Computing*, 2009.

[28] Maxima, a Computer Algebra System. [Online]. Available: http://maxima.sourceforge.net/

[29] J. A. Nelder and R. Mead, "A simplex method for function minimization," *Computer Journal*, vol. 7, p. 308313, 1965.

[30] A. Ganapathi, K. Datta, O. Fox, and D. Patterson, "A Case for Machine Learning to Optimize Multicore Performance," in *First USENIX Workshop on Hot Topics in Parallelism (HotPar '09)*, 2009.

[31] P. Conway, N. Kalyanasundharam, G. Donley, K. Lepak, and B. Hughes, "Cache Hierarchy and Memory Subsystem of the AMD Opteron Processor," *IEEE Micro*, vol. 30, pp. 16–29, 2010.