CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. 0000; **00**:1–23
Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/cpe

Autotuning divide-and-conquer stencil computations

Ekanathan Palamadai Natarajan[1*], Maryam Mehri Dehnavi[1*†], and Charles E. Leiserson[1*]

[1] *MIT Computer Science and Artificial Intelligence Laboratory, 32 Vassar Street, Cambridge, MA 02139*

SUMMARY

This paper explores autotuning strategies for serial divide-and-conquer stencil computations, comparing the efficacy of traditional "heuristic" autotuning with that of "pruned-exhaustive" autotuning. We present a pruned-exhaustive autotuner called Ztune that searches for optimal divide-and-conquer trees for stencil computations. Ztune uses three pruning properties — space-time equivalence, divide subsumption, and favored dimension — that greatly reduce the size of the search domain without significantly sacrificing the quality of the autotuned code.
We compared the performance of Ztune with that of a state-of-the-art heuristic autotuner called OpenTuner in tuning the divide-and-conquer algorithm used in Pochoir stencil compiler. Over a nightly run on ten application benchmarks across two machines with different hardware configurations, the Ztuned code ran 5%–12% faster on average, and the OpenTuner tuned code ran from 9% slower to 2% faster on average, than Pochoir's default code. In the best case, the Ztuned code ran 40% faster, and the OpenTuner tuned code ran 33% faster than Pochoir's code. Whereas the autotuning time of Ztune for each benchmark could be measured in minutes, to achieve comparable results, the autotuning time of OpenTuner was typically measured in hours or days. Surprisingly, for some benchmarks, Ztune actually autotuned faster than the time it takes to perform the stencil computation once. Copyright © 0000 John Wiley & Sons, Ltd.

Received ...

KEY WORDS: Autotuning, Stencil computations, Divide-and-conquer, Trapezoidal decomposition

1. INTRODUCTION

Many compute-intensive scientific applications perform "stencil computations." A ***stencil*** defines the value of a grid point in a d-dimensional spatial grid at time t as a function of the values of neighboring grid points at recent times before t. A ***stencil computation*** [1–18] involves computing the stencil at each grid point for several time steps. Stencil computations are conceptually simple to implement using nested loops, but naive looping implementations suffer from poor cache performance. ***Tiling*** [2, 19, 20] can enhance performance, although tiling can make the code overly specific to a particular cache size, decreasing portability across machines. Cache-oblivious [21] divide-and-conquer stencil codes based on Frigo and Strumpen's trapezoidal decomposition [5, 6] are robust to changes in cache size and provide an asymptotic improvement in cache efficiency over looping implementations.

Because stencil computations constitute a dominant part of the compute time of many important scientific applications, the problem of ***autotuning*** [2, 9, 22–30] — automatically selecting key

†Author's current address is Rutgers University, Department of Electrical and Computer Engineering, 94 Brett Rd, Piscataway, NJ 08854, and the current email address is maryam.mehri@rutgers.edu
*Correspondence to: MIT Computer Science and Artificial Intelligence Laboratory, 32 Vassar Street, Cambridge, MA 02139. E-mail: {epn, mmehri, cel}@mit.edu

Copyright © 0000 John Wiley & Sons, Ltd.

constants in an application to optimize its performance — has emerged as an important technology for stencil computations (see, for example, [2,9,26,27]). Autotuning is especially valuable whenever a stencil code must be ported to a machine with a different hardware configuration from where the code was originally developed. Without proper tuning, significant performance can be lost. Because the original programmer may no longer be around to tune the adjustable constants in the code, it makes sense to automate the tuning process.

Historically, the earliest application-specific autotuners were ***exhaustive***: they simply enumerated the search domain.[†] Examples for exhaustive autotuners include FFTW [22, 23], which optimizes Fourier transforms, and ATLAS [24], which optimizes matrix multiplication. Since exhaustive enumeration can be time consuming for some applications, later autotuners [2, 9, 25–30] were ***heuristic***, using heuristics and sophisticated machine-learning methods to find parameter settings that produced good runtimes. There are several prominent examples for application-specific heuristic autotuners. PATUS [26] and Sepya [27] autotune tiled stencil computations using machine-learning based search techniques. SPIRAL [28] uses heuristic search to generate platform-tuned implementations for digital signal processing algorithms. OSKI [25] autotunes sparse matrix kernels using both heuristic and exhaustive search methods. Other heuristic autotuning packages include PetaBricks [29], which autotunes algorithmic choices in programs, and Active Harmony [30], which is an automated runtime tuning system. More recently, heuristic autotuning frameworks, such as OpenTuner [31], have allowed programmers to build application-specific autotuners. Despite this recent trend towards heuristic autotuning, this paper examines if exhaustive autotuning with pruning is viable for the domain of divide-and-conquer stencil computations. It describes a pruned-exhaustive autotuning strategy, and compares its efficacy with state-of-the-art heuristic autotuning in optimizing the performance of a serial divide-and-conquer stencil code based on the TRAP algorithm used in the Pochoir stencil compiler [15].

Given the attention that "parallelism" receives in high-performance computing, why optimize the performance of "serial" divide-and-conquer stencil codes? Of course, it would be ideal to optimize the performance of parallel divide-and-conquer stencil codes using autotuning. However, as a first step in that direction, it is useful to understand how to autotune serial divide-and-conquer stencil codes and the performance improvements accrued in autotuning such codes, before autotuning their parallel counterparts. Serial codes are relatively simple to autotune since they do not face a host of issues that arise in a parallel execution like memory bandwidth saturation, interprocessor communication, and nondeterminism in scheduling work among processors. Hence it makes sense to develop the theory and practice of autotuning the serial execution of divide-and-conquer stencil codes, and then extend the theory and practice to autotune their parallel execution taking into account the additional issues that arise in a parallel execution. Besides, optimized serial codes tend to be competitive with their parallel counterparts, and often motivate efficient parallel implementations. For example, we compared the performance of the autotuned serial TRAP algorithm with that of Pochoir's parallel TRAP algorithm in computing a periodic heat [32] stencil on a 2D grid of size 4096×4096 for 512 timesteps. On a modern Ivy Bridge machine, whose specs are shown in Figure 19, the autotuned serial TRAP code was just 16% slower than the parallel TRAP code executed using 2 processor cores. Of course, with more cores, the parallel execution will be much faster. Nevertheless, parallel codes are typically benchmarked against optimized serial codes.

Since our focus is on autotuning serial codes, we modified TRAP to disable parallelism. We also disabled the "hyperspace cuts," which enhance the parallelism of TRAP, and replaced them with sequential cuts, as in the original algorithms due to Frigo and Strumpen [5, 6]. The modified code performs equivalently to Pochoir's original TRAP code when run serially. For the rest of the paper, we will use the term TRAP to refer to the serial version of the TRAP code.

At a higher level, the TRAP code executes a fixed recursion, dividing a given problem into subproblems, or terminating the recursion and executing a base case kernel function when the size of the problem falls beneath a threshold constant. The actual code is more complicated in that it

[†]We use the term ***search domain*** rather than the more conventional ***search space*** to avoid confusion with the geometric use of "space" endemic to stencil computations.

Copyright © 0000 John Wiley & Sons, Ltd.

Prepared using cpeauth.cls
DOI: 10.1002/cpe

				Nehalem		Ivy Bridge	
Benchmark	*Dims*	*Grid size*	*Time steps*	*OpenTuner*	*Ztune*	*OpenTuner*	*Ztune*
APOP	1	$2,000,000$	524,288	0.98	0.98	1.32	0.96
Heat1	2	1000×2000	512	0.93	0.94	0.92	0.92
Heat2	2p	$100 \times 20,000$	8192	0.84	0.88	0.84	0.89
Life	2p	2000×3000	1024	0.97	0.97	0.99	0.99
Heat3	2p	4000×4000	1024	0.94	0.96	0.89	0.91
Heat4	2p	4096×4096	512	1.00	1.00	0.67	0.60
Heat5	2p	$10,000 \times 10,000$	4096	1.16	0.92	1.00	0.87
LBM	3	$100 \times 100 \times 130$	64	0.84	0.98	0.93	0.97
Wave	3	$1000 \times 1000 \times 1000$	64	3.57	0.96	1.45	0.88
Heat6	4	$70 \times 70 \times 70 \times 70$	32	0.98	0.90	0.99	0.87
Geometric mean				1.09	0.95	0.98	0.88

Figure 1: Performance of autotuned TRAP codes relative to Pochoir's default hand-tuned TRAP code on ten stencil benchmarks. The reported numbers are the better of two runs. OpenTuner was run 16 hours to autotune each benchmark, whereas Ztune took its natural time: less than an hour for Wave and at most a few minutes for the others. The header *Dims* indicates the number of spatial dimensions of the grid. The stencils are periodic if the *Dims* column contains a "p". The header *Grid size* indicates the sizes of the spatial dimensions of the grid. The header *Time steps* indicates the size of the time dimension. The benchmarks are sorted first by the number of spatial dimensions and second by spatial volume. The header *OpenTuner* indicates the ratio of the runtime of OpenTuner tuned TRAP code to the runtime of Pochoir's TRAP code. The header *Ztune* indicates the ratio of the runtime of Ztuned TRAP code to the runtime of Pochoir's TRAP code. A lower ratio indicates that the autotuned code runs faster than Pochoir's code. The last row displays the geometric means of the ratios. The header *Nehalem* gives ratios on the machine, on which Pochoir was developed. The header *Ivy Bridge* gives ratios on a relatively modern machine that supports AVX instructions. The detailed specs of the machines are shown in Figure 19.

uses two different kernel functions, a faster kernel at the interior of the grid, and a slower kernel at the boundaries of the grid that checks if memory accesses fall off the grid. Hence, the code uses two different threshold constants, one for problems that lie completely in the interior, and the other for problems that impinge on the boundary. These threshold constants along different "dimensions" of the grid were determined by trial-and-error hand tuning by the authors of Pochoir. Handtuning the values for the threshold constants helps avoid recursive function-call overhead, optimally use the computer's memory hierarchy, and handle stencil computations at the boundaries efficiently.

The traditional way of autotuning TRAP is to simply parameterize the threshold constants, and to search for optimal parameter values over the domain of possible base-case sizes called the ***base domain***. In contrast, this paper presents a different autotuning strategy that gains insight from how TRAP operates. We define a search domain called the ***choice domain*** that generalizes TRAP, where at each problem in the recursion tree, a parameter is created that chooses whether to divide the problem in one of several ways and recur, or execute the base case. As we shall see, the choice domain is much larger in size than the base domain, but can be pruned to produce faster autotuning times. We present an exhaustive autotuner called ***Ztune*** that searches the choice domain to find the fastest recursion tree for a stencil computation. Although Ztune, in principle, exhaustively enumerates the choice domain, it uses three pruning properties that prune the choice domain effectively. These properties reduce the autotuning time by orders of magnitude without significantly sacrificing the performance of the tuned stencil code.

The sheer number of choice parameters in the choice domain renders naive heuristic search over that domain infeasible. Consequently we had to resort to heuristic search over the base domain. To that effect, we used the state-of-the-art OpenTuner framework [31] to perform heuristic search over the the base domain, and find optimal values for the threshold constants. OpenTuner takes as input, the parameterized threshold constants and the base domain specification for each parameter. It then finds optimal values for these parameters by running the TRAP code with different parameter settings from the base domain, while simultaneously pruning the base domain using heuristics and machine-learning based algorithms.

Copyright © 0000 John Wiley & Sons, Ltd.
Prepared using cpeauth.cls

Figure 1 compares the performance of autotuned TRAP codes with that of Pochoir's default TRAP code using the hand-tuned threshold constants, on ten stencil benchmarks on two machines, whose specifications are shown in Figure 19. The benchmark suite includes American put stock-option pricing (APOP) [33]; a heat equation [32] on a 2D grid (Heat1), a 2D torus (Heat2, Heat3, Heat4, and Heat5), and a 4D grid (Heat6); Conway's game of Life (Life) [34]; the lattice Boltzmann method (LBM) [35]; and a 3D finite-difference wave equation (Wave) [36]. For this experiment, we ran Ztune until it completed, which took less than an hour for any benchmark, and typically less than 7 minutes. Since OpenTuner took several days to completion, we ran OpenTuner for 16 hours (a nightly run) and recorded the results.

Compared with Pochoir's hand-optimized code across the two machines, the Ztuned code is on average[‡] 5%–12% faster, and the OpenTuner tuned code is on average 9% slower to 2% faster. On the Nehalem machine, where Pochoir was developed, the Ztuned code is on average 5% faster. Supporting the contention that tuning attains even more importance when porting code, on the more modern Ivy Bridge machine, which has a different hardware configuration from Nehalem, the Ztuned code is on average 12% faster. Moreover, Ztune generally produces a faster tuned code than OpenTuner. In particular, the Ztuned code is on average 11%–15% faster than the OpenTuner tuned code across the two machines.

The scenario where a scientist repeats the stencil computation on the same grid several times is common. Since autotuning time can be amortized over several runs of stencil computation, it makes sense to autotune the stencil code for a given grid and for a given number of time steps. For cases where the number of time steps is significantly larger than the spatial dimensions of the grid, the pruning properties reduce the tuning problem in practice to one where the number of time steps is proportional to the spatial dimensions, thereby saving considerable autotuning time. Even for our benchmarks, where the spatial and time dimensions are proportional to each other, the autotuning time of Ztune is at most a few multiples of the runtime of the tuned code, and, surprisingly, sometimes the autotuning time is less than the runtime.

Contributions

This paper makes the following research contributions:

- We describe Ztune, an exhaustive autotuner for serial divide-and-conquer stencil computations.
- We use three ***pruning properties*** — space-time equivalence, divide subsumption, and favored dimension — which improve the runtime of Ztune significantly, and we document the advantage that each property accrues to Ztune.
- We show that the memory consumption of Ztune under space-time equivalence is $O(2^d n \lg h)$, where n is the spatial volume of a d-dimensional spatial grid and h is the size of the time dimension. We also show that Ztune takes $O(2^d n^2 h)$ time to autotune under space-time equivalence.
- We demonstrate empirically that Ztune can produce faster divide-and-conquer stencil codes with less tuning time than state-of-the-art heuristic autotuning.

Outline

The remainder of the paper is organized as follows. Section 2 reviews the serial TRAP algorithm and the "planned" TRAPPLE algorithm, which is optimized by Ztune. Section 3 describes the Ztune algorithm. Section 4 provides an overview of the three pruning properties, reports on the performance of Ztune on varying grid sizes, and discusses the effects of noise in autotuning. Sections 5, 6, and 7 describe each of the pruning properties in detail. Section 8 describes OpenTuner, and reports on relative comparisons between Ztuned and OpenTuner tuned codes, when OpenTuner is run a few multiples of time longer than Ztune. Besides, Section 8 also compares the performance

[‡]All averages of ratios in this paper are geometric means.

Copyright © 0000 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe

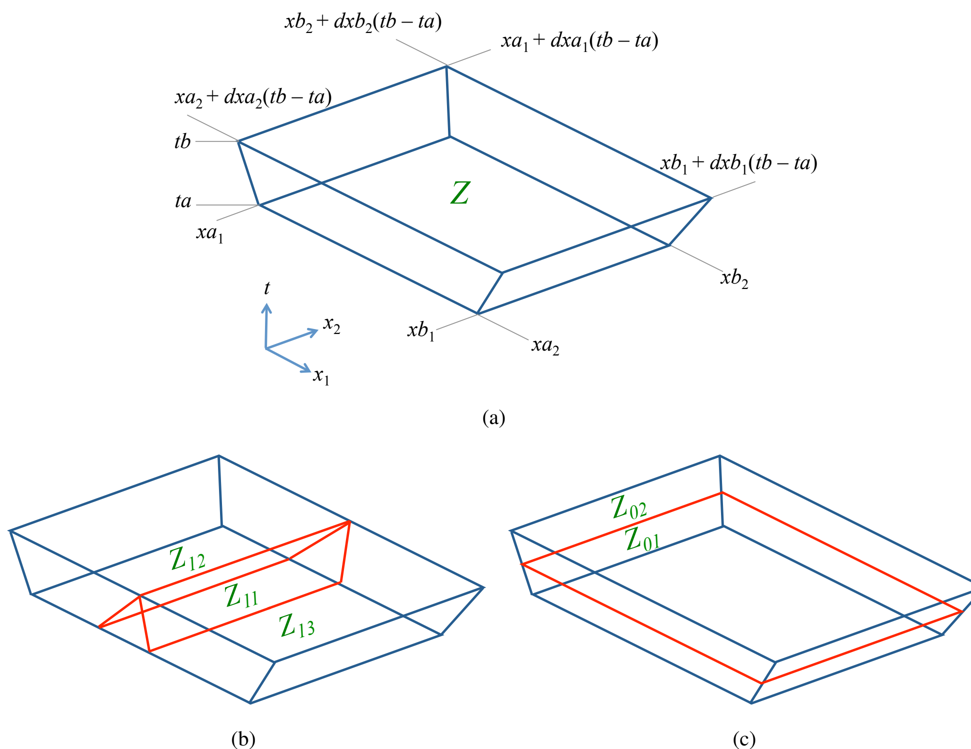

Figure 2: Dividing a 3-dimensional zoid with space and time cuts. (a) A zoid $Z = \text{zoid}(ta, tb, xa_1, xb_1, dxa_1, dxb_1, xa_2, xb_2, dxa_2, dxb_2)$. (b) An x_1-cut of Z trisects the zoid into three side-by-side subzoids Z_{11}, Z_{12} and Z_{13}. The stencil dependencies require that Z_{11} be processed before Z_{12} and Z_{13}. (c) A time cut of Z bisects the zoid into bottom and top subzoids Z_{01} and Z_{02}, respectively. The stencil dependencies require that Z_{01} be processed before Z_{02}.

of Ztuned divide-and-conquer stencil codes with that of autotuned tiling-based stencil codes, and reports on the performance of Ztune on two more machines with different architectures. Section 9 offers some concluding remarks.

2. THE TRAP AND TRAPPLE ALGORITHMS

This section describes the TRAP algorithm used by the Pochoir stencil compiler [15], and how we can adapt it into a "planned" algorithm called TRAPPLE suitable for autotuning by Ztune. This section also defines a "plan-finding" problem for autotuning divide-and-conquer stencil codes.

Although the details of TRAP are generally unimportant for the discussion of autotuning, it is helpful to understand a little about it. TRAP operates recursively on regions of a space-time grid called ***hypertrapezoids***, or simply ***zoids*** [15] for short. A 3-dimensional zoid $Z \subseteq \mathbf{N} \times \mathbf{Z}^2$, can be specified by two time coordinates $ta, tb \in \mathbf{N}$, two x_1-coordinates $xa_1, xb_1 \in \mathbf{Z}$, two x_1-slopes[§] $dxa_1, dxb_1 \in \mathbf{Z}$, two x_2-coordinates $xa_2, xb_2 \in \mathbf{Z}$, and two x_2-slopes $dxa_2, dxb_2 \in \mathbf{Z}$. The zoid defined

[§]Technically, inverse slopes, but we follow the terminology of [5].

Copyright © 0000 John Wiley & Sons, Ltd.
Prepared using cpeauth.cls

```
TRAP(Z)
 1  h = height(Z)
 2  if h < thresh0 // base case
 3      for each (t, x1, x2) ∈ Z
 4          KERNEL(t, x1, x2)
 5  else  // recursive case
 6      w1 = width(Z, 1)
 7      w2 = width(Z, 2)
 8      if w1 ≥ max(thresh1, 4σ1h) // x1-cut
 9          trisect Z with an x1-cut into subzoids Z11, Z12, and Z13
10          TRAP(Z11) ; TRAP(Z12) ; TRAP(Z13)
11      elseif w2 ≥ max(thresh2, 4σ2h) // x2-cut
12          trisect Z with an x2-cut into subzoids Z21, Z22, and Z23
13          TRAP(Z21) ; TRAP(Z22) ; TRAP(Z23)
14      else // time cut
15          bisect Z with a time cut into subzoids Z01 and Z02
16          TRAP(Z01) ; TRAP(Z02)
```

Figure 3: Pseudocode for the TRAP divide-and-conquer stencil algorithm operating on a 3-dimensional zoid. TRAP takes as input a 3-dimensional zoid Z. The values $thresh_0$, $thresh_1$, and $thresh_2$ are threshold constants, which can be tuned. The values σ_1 and σ_2 are the slopes [15] of the stencil in the x_1- and x_2-dimensions, respectively. The application-specific KERNEL procedure is not shown.

by these parameters is given by

$$\begin{aligned} Z &= \text{zoid}(ta, tb, xa_1, xb_1, dxa_1, dxb_1, xa_2, xb_2, dxa_2, dxb_2) \\ &= \{(t, x_1, x_2) \in \mathbf{N} \times \mathbf{Z}^2 : ta \le t < tb \; ; \\ &\qquad xa_1 + dxa_1(t - ta) \le x_1 < xb_1 + dxb_1(t - ta) \; ; \text{and} \\ &\qquad xa_2 + dxa_2(t - ta) \le x_2 < xb_2 + dxb_2(t - ta)\} \; . \end{aligned}$$

This definition can be straightforwardly extended to a $(d+1)$-dimensional zoid spanning d spatial dimensions and the time dimension. As an example, a 3-dimensional zoid Z is shown in Figure 2(a). The ***height*** of Z is given by $\text{height}(Z) = tb - ta$. The ***width*** of Z along the x_1-dimension is the maximum of its two lengths along the x_1-dimension, that is, $\text{width}(Z, 1) = max(xb_1 - xa_1, (xb_1 + dxb_1(tb - ta)) - (xa_1 + dxa_1(tb - ta)))$. The width along the x_2-dimension is defined similarly. The ***spatial volume*** of Z is the product of its widths along all spatial dimensions.

The basic algorithm

Figure 3 shows the pseudocode for TRAP operating on a 3-dimensional zoid. For didactic purposes, we have abstracted away many details, which are well described in [5,6,15]. TRAP works as follows. In the base case, which occurs when the height of the zoid falls below a threshold $thresh_0$ tested for in line 2, lines 3–4 perform the stencil computation on each grid point in the input zoid Z using an application-specific KERNEL procedure. In the recursive case, lines 8–16 perform either a ***space cut*** of Z along one of the spatial dimensions or a ***time cut*** of Z along the time dimension, and make recursive calls on the subzoids. An x_1-cut and a time cut on a 3-dimensional zoid are illustrated in Figures 2(b) and 2(c), respectively. Note that Frigo and Strumpen's original stencil algorithm [5] bisects both in space and time. Since we are autotuning the TRAP code in Pochoir, we follow Pochoir's conventions of trisecting in space and bisecting in time.

TRAP makes specific ***divide choices*** at each recursive step. It may choose to execute the base case (lines 3–4) or divide the given zoid along the x_1 dimension (lines 9–10), or the x_2 dimension (lines 12–13), or the time dimension (lines 15–16) into subzoids. As shown in Figure 3, the divide choices are influenced by different threshold constants, which were determined by trial-and-error

Copyright © 0000 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe

```
TRAPPLE(Z, z)
 1  if z.choice = −1 // base case
 2      for each (t, x1, x2) ∈ Z
 3          KERNEL(t, x1, x2)
 4  else // recursive case
 5      if z.choice = 1 // x1-cut
 6          trisect Z with an x1-cut into subzoids Z11, Z12, and Z13
 7          let z.children = ⟨z11, z12, z13⟩
 8          TRAPPLE(Z11, z11) ; TRAPPLE(Z12, z12) ; TRAPPLE(Z13, z13)
 9      elseif z.choice = 2 // x2-cut
10          trisect Z with an x2-cut into subzoids Z21, Z22, and Z23
11          let z.children = ⟨z21, z22, z23⟩
12          TRAPPLE(Z21, z21) ; TRAPPLE(Z22, z22) ; TRAPPLE(Z23, z23)
13      elseif z.choice = 0 // time cut
14          bisect Z with a time cut into subzoids Z01 and Z02
15          let z.children = ⟨z01, z02⟩
16          TRAPPLE(Z01, z01) ; TRAPPLE(Z02, z02)
17      else error "invalid choice"
```

Figure 4: Pseudocode for the planned algorithm TRAPPLE operating on a 3-dimensional zoid Z. In addition to Z, TRAPPLE takes as input the root node z of a "plan" for Z. TRAPPLE assumes that the children of z and the subzoids of Z are maintained in the same order.

handtuning by the authors of Pochoir. Different values can yield different runtimes, and handtuning to find optimal values that minimize runtime can be a laborious task.

For example, suppose that TRAP is passed a zoid Z that fits in the computer's L1 cache, and that height$(Z) \geq 2$. If the threshold constant $thresh_0$ is set to 2, then TRAP would divide the zoid and recur. But terminating the recursion and executing the base case might be faster than dividing and recurring, because it avoids function-call overhead. On the other hand, if $thresh_0$ is set to too large a value, and if Z does not fit in L1, then executing the base case might result in many cache misses, yielding slower code. Thus, $thresh_0$ must be tuned, as must the other tunable constants. The actual TRAP code in Pochoir is more complicated than that shown in Figure 3, and uses two kernel functions: a faster kernel at the interior of the grid, and a slower kernel at the boundaries that checks if memory accesses fall off the grid. Hence, two different threshold constants are used in each dimension, one for zoids that lie completely in the interior, and the other for zoids that impinge on the boundary. It might be better to divide zoids that impinge on the boundary, even if they fit in cache, if the subzoids use more of the faster kernel. The tradeoff between tuning for cache locality and for faster kernel usage makes handtuning the threshold constants even harder.

The planned algorithm

The tuning strategy employed by Ztune is not to choose values for the tunable constants like $thresh_0$. Instead, Ztune focuses on the divide choices themselves, associating each recursive instantiation of the TRAP function with its own divide choice. Given a $(d+1)$-dimensional zoid Z, each instantiation of TRAP has at most $d+2$ divide choices: it can make a time cut; it can make an x_i-cut for some $i \in \{1,2,\ldots,d\}$; or it can terminate the recursion and execute the base case. We shall represent the divide choice for Z as an integer in the set $\{-1,0,1,\ldots,d\}$, where -1 corresponds to the base case, 0 corresponds to the time dimension, and choices $1,2,\ldots,d$ correspond to the spatial dimensions $x_1,x_2,\ldots,x_d$, respectively.

Of course, we must modify TRAP so that it is parametrized by the divide choices rather than the threshold constants. Figure 4 shows how TRAP can be transformed into a ***planned algorithm*** called TRAPPLE, which uses divide choices for making decisions about the divide-and-conquer execution of the code. TRAPPLE operates on a ***plan***, which is an ordered tree of divide choices made by the algorithm at each recursive step of computation. Each node z in the plan correponds to a zoid Z that arises during the execution of TRAPPLE, and has the following fields:

Copyright © 0000 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe

- $z.choice$: the divide choice for Z, drawn from $\{-1, 0, 1, \ldots, d\}$.
- $z.children$: an ordered list of the children of z, or NIL if the divide choice is -1, corresponding to a base case.

If the divide choice for a node z, which corresponds to a zoid Z, is to divide Z into k subzoids $Z_1, Z_2, \ldots, Z_k$, then z has k ordered children $z_1, z_2, \ldots, z_k$, where node z_i corresponds to zoid Z_i. As TRAPPLE executes, it consults the root node z of the plan for the input zoid Z. Using the divide choice stored in $z.choice$, it either executes the base case, or divides Z into subzoids and recursively calls itself on each, passing the subzoid Z_i and the corresponding child node z_i as arguments.

Plan-finding

The autotuning problem for divide-and-conquer stencil codes can now be stated as follows. Define the ***(execution) cost*** of a piece of code to be the time taken to execute the code on a given computer. The ***cost*** of a plan is the time taken to execute the plan. The ***plan-finding*** problem is to find an ***optimal plan***, that is, a plan with the minimum cost, to perform stencil computation on a given grid on a given computer. At its heart, Ztune is simply a plan-finding algorithm for TRAPPLE. Two computers may have different costs for the same plan, due to differences in their hardware architecture, operating system, and compiler, among other things. For example, the cost of a plan may vary with the cache size. Hence, a solution to the plan-finding problem may not be portable across computers.

3. ZTUNE

This section presents the Ztune algorithm, which finds an optimal plan for the TRAPPLE stencil algorithm. Ztune can be slow, but later sections will show how to make it run fast. This section also formalizes the search domain explored by Ztune.

Figure 5 shows the pseudocode for procedure ZTUNE, which instruments the TRAPPLE algorithm with timer calls, and finds an optimal plan for stencil computation on a $(d+1)$-dimensional zoid Z. Although performance is not exactly repeatable, our experience with Ztune shows that instrumentation can be sufficiently accurate to produce highly efficient plans. ZTUNE uses the following auxiliary procedures:

- TIC() starts a timer.
- TOC() stops the timer and returns the elapsed time since the last call to TIC.
- CHOICE(Z) returns the set of possible divide choices for Z, where the set excludes the choice -1 for executing Z as a base case.
- BASE-CASE(Z, z), shown in Figure 6, chooses base case as the divide choice for Z, if executing the base case is no more expensive than the current plan for Z.

Besides ZTUNE uses two procedures INSERT and LOOKUP, which store and look up optimal plans for zoids, avoiding repeated plan-finding for the same zoid. We shall discuss the implementation of INSERT and LOOKUP in Section 5, where we introduce the notion of equivalence among zoids. ZTUNE measures the costs of all possible divide choices for zoid Z, and stores the best choice in the corresponding node z. Each node z in the plan has an additional field $z.cost$, which stores the ***optimal cost***, that is, the cost of an optimal plan for the corresponding zoid. Determining where to place TIC and TOC within code in order to properly measure all relevant costs without also capturing bookkeeping overhead is a bit tricky. To measure function-call overhead in calling ZTUNE, a call to ZTUNE must be immediately preceded by a TIC and immediately succeeded by a TOC.

ZTUNE works as follows. Line 1 measures the ***call cost***, that is, the function-call overhead in invoking ZTUNE. Line 23 starts the timer, which the caller of ZTUNE can stop and measure the ***return cost***, that is, the function-call overhead in returning from ZTUNE. The sum of the call and return costs is the ***link cost***, which is the total function-call overhead in calling ZTUNE on Z. If LOOKUP finds the root node z of an optimal plan for zoid Z in line 2, then ZTUNE simply returns z in line 24. Otherwise, ZTUNE proceeds as follows.

Copyright © 0000 John Wiley & Sons, Ltd.

Prepared using cpeauth.cls
DOI: 10.1002/cpe

```
ZTUNE(Z)
1   call-cost = TOC()
2   z = LOOKUP(Z)
3   if z == NIL
4       Allocate plan node z
5       z.cost = ∞
6       z.children = NIL
7       C = CHOICE(Z)
8       for each choice c ∈ C
9           TIC()
10          Divide Z using choice c into k_c subzoids Z_c1, Z_c2, ..., Z_ck_c
11          rec-cost = TOC()
12          for i = 1 to k_c
13              TIC()
14              (z_ci, call-cost_ci) = ZTUNE(Z_ci)
15              ret-cost_ci = TOC()
16              rec-cost += call-cost_ci + z_ci.cost + ret-cost_ci
17          if rec-cost < z.cost
18              z.cost = rec-cost
19              z.choice = c
20              z.children = ⟨z_c1, z_c2, ..., z_ck_c⟩
21      BASE-CASE(Z, z)
22      INSERT(Z, z)
23  TIC()
24  return (z, call-cost)
```

Figure 5: Procedure ZTUNE, which finds an optimal plan for the TRAPPLE stencil algorithm operating on a $(d+1)$-dimensional zoid Z. ZTUNE returns an ordered pair consisting of the root node z of the optimal plan and the partial function-call overhead *call-cost*. The sections of code that are timed are highlighted in blue and underlined. Although line 14 is shown as being timed, only the function-call overhead in calling ZTUNE is measured. The definitions of the TIC, TOC, CHOICE, INSERT, and LOOKUP procedures are not shown.

```
BASE-CASE(Z, z)
1   TIC()
2   for each (t, x_1, x_2, ..., x_d) ∈ Z
3       KERNEL(t, x_1, x_2, ..., x_d)
4   base-cost = TOC()
5   if base-cost ≤ z.cost
6       z.cost = base-cost
7       z.choice = −1 // base case
8       z.children = NIL
```

Figure 6: Pseudocode to execute the base case. BASE-CASE takes as input a zoid Z and the root node z of a plan for Z. The sections of code that are timed are highlighted in blue and underlined.

Recursive case For each possible divide choice c of zoid Z, lines 9–11 measure the ***divide cost*** of Z, that is, the time taken to divide Z using choice c into subzoids. The ***recursion cost*** of Z for choice c is the sum of the divide cost of Z, and the link and optimal costs of subzoids created using choice c. Lines 11–16 compute the recursion cost for choice c, which is maintained in variable *rec-cost*, as follows. Line 11 initializes *rec-cost* with the divide cost of Z. Line 14 recursively finds an optimal plan for subzoid Z_{ci} and the call cost. Line 15 measures the return cost. Line 16 increments *rec-cost* with the sum of the link and optimal costs of Z_{ci}. Lines 18–20 update the attributes of node z, if *rec-cost* is smaller than the cost of the current plan for Z.

Base case Line 21 invokes procedure BASE-CASE, which is shown in Figure 6. BASE-CASE works as follows. Lines 1–4 measure the ***base-case cost*** of Z, that is, the time to perform stencil

Copyright © 0000 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe

computation on Z using an application-specific procedure KERNEL. If the measured base-case cost is at most the cost of the current plan for Z, then lines 6–8 update the attributes of node z accordingly.

Line 22 of ZTUNE invokes INSERT, which inserts node z, as the root of an optimal plan for zoid Z, into a lookup table. Procedure ZTUNE is guaranteed to terminate, if TRAPPLE terminates for each divide choice of Z.

Search domain

In principle, the basic Ztune algorithm explores the search domain of all possible ways of executing the planned divide-and-conquer stencil algorithm TRAPPLE in order to select the best plan. To close this section, we formalize this notion.

The ***choice domain*** for TRAPPLE run on a given $(d+1)$-dimensional zoid Z can be viewed as the set of all possible plans for Z. Many of the plans have a common structure, and in particular, the roots of all plans share one of the at most $d+1$ divide choices that actually divide Z, as well as the -1 base-case divide choice. We can represent the entire choice domain as a single tree as follows. For any node z in the tree corresponding to a zoid Z, let k_c be the number of subzoids produced by choice $c \in \{0,1,\ldots,d\}$. Then node z has $k = \sum_{c \in \text{CHOICE}(Z)} k_c$ children, where the child z_{ci} corresponds to the ith zoid produced by dividing Z according to choice c. Since each node in the choice domain corresponds to a particular zoid, for convenience, we shall sometimes view the choice domain as this tree of zoids. ZTUNE can now be viewed as walking this search domain tree, finding the optimal cost at each node in the tree, selecting the best divide choice, and recording the results in a plan, which is a subtree of the choice domain.

Analysis

The number of plans in the choice domain is huge, however, and is exponential in the height of the zoid at the root, as we shall see in the following analysis. First, we state the following lemma, which we use in our analysis, and skip its proof.

Lemma 1
Consider the recursion tree R obtained by dividing an integer $m > 0$ into $\lfloor m/2 \rfloor$ and $\lceil m/2 \rceil$, and recursively dividing those two integers similarly, until we have 1. There exist at most 2 distinct integers $\lfloor m/2^k \rfloor$ and $\lceil m/2^k \rceil$ at level $k \geq 0$ of tree R. □

Since the recursion tree R has $\lg m + 1$ levels, it follows from Lemma 1 that R has at most $2\lg m + 2$ distinct integers.

Theorem 2
Consider a zoid Z with height $h > 1, h \in \mathbf{Z}^+$. The number of plans in the choice domain rooted at Z is $\Omega(2^{h/2})$.

Proof
At each time cut, the TRAPPLE algorithm bisects a zoid with height $h > 1$ into two subzoids of heights $\lfloor h/2 \rfloor$ and $\lceil h/2 \rceil$. It follows from Lemma 1 that the set S of heights of zoids in the choice domain is given by $\{h, \lfloor h/2 \rfloor, \lceil h/2 \rceil, \ldots, 1\}$. We use induction over the heights in S to prove the theorem. Let $P(h)$ be the number of plans in the choice domain rooted at a zoid with height h. Then the inductive hypothesis is given by $P(h) \geq 2^{h/2}$. We consider 2 base cases, namely heights 2 and 3, as possible values for the heights $\lfloor h/2^k \rfloor, \lceil h/2^k \rceil \in S$ for some $k \in \mathbf{Z}^+$. Note that recursive time cuts of a zoid with height $h > 3$ will eventually result in subzoids with height 2 or 3. The base case holds for $h = 2$, since a height 2 zoid can be divided in time, or executed as a base case. Similarly, a height 3 zoid can be divided in time into subzoids of heights 1 and 2, or executed as a base case, producing $P(3) = 3 \geq 2^{3/2}$ different plans. For the inductive step, assume that $P(\lfloor h/2 \rfloor) \geq 2^{\lfloor h/2 \rfloor /2}$, and $P(\lceil h/2 \rceil) \geq 2^{\lceil h/2 \rceil /2}$. Zoid Z with height h can be divided in time into two subzoids of heights $\lfloor h/2 \rfloor$ and $\lceil h/2 \rceil$, or executed as a base case. Then the number of plans in the choice domain rooted at Z is $P(h) = P(\lfloor h/2 \rfloor) \cdot P(\lceil h/2 \rceil) + 1 \geq 2^{h/2}$. □

Copyright © 0000 John Wiley & Sons, Ltd.
Prepared using cpeauth.cls

DOI: 10.1002/cpe

Benchmark	*STE + DS + FD*	*STE + DS*	*STE*
APOP	1,252.39	1,251.87	—
Heat1	1.73	1.61	1.62
Heat2	51.27	51.36	51.58
Life	17.14	16.59	—
Heat3	25.32	25.19	—
Heat4	20.24	19.95	—
Heat5	576.83	582.84	—
LBM	13.23	13.42	13.48
Wave	818.74	827.87	—
Heat6	19.08	18.90	—

Figure 7: Runtimes (in seconds) of the tuned TRAPPLE code on *Ivy Bridge* under different combinations of the three pruning properties. A dash sign (—) indicates that plan-finding timed out after two days. The reported runtimes are the median of five runs.

Benchmark	*STE + DS + FD*	*STE + DS*	*STE*
APOP	36	35	—
Heat1	19	51	11,584
Heat2	234	348	85,013
Life	267	520	—
Heat3	59	202	—
Heat4	43	134	—
Heat5	314	1,116	—
LBM	34	683	103,221
Wave	3,582	53,349	—
Heat6	348	34,682	—

Figure 8: Plan-finding times (in seconds) of Ztune on *Ivy Bridge* under different combinations of the three pruning properties. A dash sign (—) indicates that plan-finding timed out after two days. The reported plan-finding times are the median of five runs.

Benchmark	*Ratio*	*Benchmark*	*Ratio*
APOP	0.03	Heat4	2.10
Heat1	11.04	Heat5	0.54
Heat2	4.56	LBM	2.54
Life	15.56	Wave	4.38
Heat3	2.32	Heat6	18.22

Figure 9: Ratio of plan-finding time to runtime under all three pruning properties. The reported ratios are the median of five runs.

The subsequent sections of the paper improve the exploration of the choice domain using three pruning properties.

4. ZTUNE'S PRUNING PROPERTIES

This section provides an overview of Ztune's pruning properties — space-time equivalence (STE), divide subsumption (DS), and favored dimension (FD) — which speed up plan-finding significantly while preserving the runtime performance of the tuned TRAPPLE code. A detailed description of the three pruning properties is deferred to Sections 5, 6, and 7. This section presents empirical results comparing autotuning times of Ztune and runtimes of the tuned TRAPPLE code under the three pruning properties. This section also describes the performance of Ztune on various grid sizes, and concludes with a discussion on the effects of noise in autotuning.

Copyright © 0000 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe

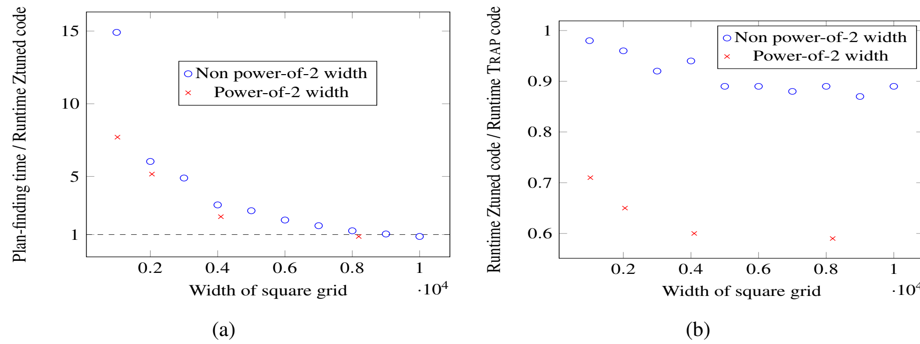

Figure 10: Performance of the Ztuned code in computing the 2D periodic heat stencil for 512 time steps on square grids with different sizes, on Ivy Bridge. The grid sizes include power-of-2 and non power-of-2 widths. (a) Ratios of plan-finding time to runtime of the Ztuned code. (b) Ratios of the runtime of Ztuned TRAPPLE code to the runtime of Pochoir's TRAP code. A lower ratio indicates that the Ztuned code runs faster than Pochoir's code. The reported ratios are the median of five runs.

The naive ZTUNE procedure is slow, because the choice domain is huge. By reducing the size of the choice domain, the three pruning properties greatly accelerate tuning. The space-time equivalence property makes the assumption that two zoids with the same shape and size have the same optimal plan, irrespective of where they are located in the space-time grid. FFTW [22, 23] uses a similar notion of equivalence in its tuning strategy. Coarsening the base-case sizes in divide-and-conquer stencil codes and tuning for tile sizes in tiling-based stencil codes implicitly assume equivalence. Because of STE, it makes sense to save plans in a lookup table so that they can be reused elsewhere in the search domain. The divide-subsumption property assumes that if a zoid is divided since it does not fit in cache, then all its ancestors in the search domain can be divided, without computing their base-case costs. The intuition is that none of the ancestors will fit in cache either, and hence measuring their base-case costs higher up in the tree is fruitless. The favored-dimension property exploits the layout of spatial grids as a linear array in memory. Computations along the unit-stride dimension execute faster than those along the other dimensions because of prefetching and vectorization. Consequently, FD restricts the search to zoids that have a longer unit-stride dimension. [37] describes experiments in tiling-based stencil codes, where choosing tiles that are longer in the unit-stride dimension results in faster runtimes.

Figure 7 shows that the various pruning properties do not significantly affect the runtimes of the tuned TRAPPLE codes. All three codes perform nearly the same for the three benchmarks all could compute. The tuned code generated under STE + DS + FD runs on average at the same speed as that generated under STE + DS. The worst-case behavior occurs in the Heat1 benchmark, where the tuned code generated under STE + DS + FD was $1.73/1.61 - 1 = 7$ percent slower.

Figure 8 shows that the pruning properties greatly accelerate plan-finding. Whereas Ztune with no pruning properties fails to find a plan for any benchmark in two days (and hence the table contains no column for no properties), STE alone succeeds in finding plans for 3 of the 10 benchmarks. STE + DS successfully finds plans for all benchmarks, speeding up plan-finding under just STE by a factor of 202 on average. Plan-finding under STE + DS + FD ran on average 4.9 times faster than plan-finding under STE + DS. The best speedup in this case occurs in the Heat6 benchmark, where plan-finding under all three properties was $34,682/348 = 99$ times faster.

Figure 9 shows the ratio of plan-finding time of Ztune to the runtime of the tuned TRAPPLE code under all three properties. This ratio is important, since it indicates how long it takes to amortize the autotuning or plan-finding time over the runtime of the tuned TRAPPLE code for a given benchmark. Interestingly, plan-finding is actually faster than the runtime for the APOP and Heat5 benchmarks. Plan-finding under all three properties, in practice, seems to take time at most a few multiples of the runtime.

Figure 10 shows the performance of Ztuned code on increasing grid sizes. Figure 10(a) shows that the ratio of plan-finding time to runtime of the tuned code decreases as grid sizes increase. Due

Copyright © 0000 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe

Benchmark	*Ratio*	*Benchmark*	*Ratio*
APOP	1.07	Heat4	1.14
Heat1	1.08	Heat5	1.09
Heat2	1.08	LBM	1.09
Life	1.00	Wave	1.03
Heat3	1.06	Heat6	0.97

Figure 11: Comparison of "predicted" and "actual" runtimes on *Ivy Bridge* under all three pruning properties. The header *Ratio* indicates the ratio of actual to predicted runtimes. Ratios close to 1.0 indicate that the predicted and actual runtimes agree, and that autotuning is less prone to noise. The reported ratios are the median of five runs.

to the pruning properties, most of the plan-finding time is spent in measuring the base-case costs of different zoids in the choice domain that fit in cache. Since the fraction of zoids that fit in cache is high for smaller grids, their plan-finding times are higher. As this fraction is low for bigger grids, their plan-finding times are also relatively lower. Figure 10(b) shows that larger grids benefit more from tuning. The Ztuned code is at most 12% faster for grids with non power-of-2 widths, and at most 40% faster for grids with power-of-2 widths.

The results from plan-finding are highly reproducible. In 10 plan-finding runs on each benchmark under all three pruning properties on the Ivy Bridge machine, the tuned runtimes differed by at most 1% on average.

Effects of noise on measurements

The accuracy of time measurements can be affected by different sources of noise. Noise during autotuning can be broadly classified into two types. Noise from sources external to the autotuner and the tuned program, such as hardware and system effects, are ***exogenous***. Noise internal to the auotuner and the tuned program, due to their implementation and the assumptions made by the pruning properties, are ***endogenous***. To keep exogenous noise minimal, we ran the experiments in a quiesced machine environment. We further disabled noise sources like Intel's Turbo Boost technology that can change the processing core's operating speed during autotuning. We took care to avoid endogenous noise sources like printf and cout statements, which interfere with measurements during tuning. The effects of ***cold misses***, which are incurred the first time data is brought into cache, on measurements are negligible since the number of cold misses is significantly smaller than the number of memory accesses for stencil codes. For example, given a zoid with spatial volume n and height h, the number of cold misses is n, whereas the number of memory accesses is $\Theta(nh)$. To keep the cold miss effects minimal, we warmed up the cache by performing stencil computation on the entire spatial grid for a few time steps before tuning.

Figure 11 gives a qualitiative measure of noise in autotuning. We define ***Predicted time*** to be the cost of an optimal plan found by plan-finding, that is, the estimated runtime of the Ztuned code. ***Actual time*** is the time taken to execute the plan, that is, the actual runtime of the Ztuned code. Figure 11 shows that the predicted and actual runtimes of the plans differ on average by 6%, and indicates that Ztune's tuning strategy is less prone to noise.

We shall examine the pruning properties in detail in the next 3 sections.

5. SPACE-TIME EQUIVALENCE PROPERTY

This section describes Ztune's "space-time equivalence" (STE) property, which speeds up plan-finding by assuming that zoids with the same shape and size have the same optimal plan irrespective of their location in the space-time. We show that plan-finding under STE incurs $O(2^d n \lg h)$ memory overhead, where n and h are the spatial volume and height, respectively, of a $(d+1)$-dimensional zoid Z. We also show that plan-finding for Z under STE takes $O(2^d n^2 h)$ time. This section concludes

Copyright © 0000 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe

with a discussion on the practical validity of STE. As mentioned before, STE succeeds in finding plans on 3 of the 10 benchmarks given two days of plan-finding time for each benchmark.

Two 3-dimensional zoids $Z \subseteq \mathbf{N} \times \mathbf{Z}^2$ and $Z' \subseteq \mathbf{N} \times \mathbf{Z}^2$ are ***space-time equivalent*** if there exists a bijection $f : Z \to Z'$ and constants $T, X_1, X_2 \in \mathbf{Z}$ such that for all $(t, x_1, x_2) \in Z$, we have $f(t, x_1, x_2) = (t + T, x_1 + X_1, x_2 + X_2)$, that is, Z' is a translation of Z in space-time. Space-time equivalent zoids in higher dimensions are defined similarly. The ***space-time equivalence*** property states that two space-time equivalent zoids have the same optimal plan. Two zoids that are not space-time equivalent are ***space-time distinct***, or ***distinct*** for short. The functionality of INSERT and LOOKUP auxiliary procedures under space-time equivalence can now be defined.

- INSERT(Z, z) inserts the root node z of an optimal plan for zoid Z into a lookup table, which maintains the root nodes of optimal plans for distinct zoids. The lookup table can be a dictionary, for example a hash table.
- LOOKUP(Z) searches the lookup table for the root node of an optimal plan for Z.

Analysis

Before we analyze space-time equivalence, it is important to note that the TRAPPLE algorithm takes $\Theta(n)$ memory and $\Theta(nh)$ time to perform stencil computation on a $(d+1)$-dimensional zoid with spatial volume n and height h. Hence, it would be useful if plan-finding does not incur significantly more memory and time overheads than TRAPPLE.

We introduce some definitions, many of which are borrowed or adapted from [5, 6, 15]. A 3-dimensional zoid $Z = \text{zoid}(ta, tb, xa_1, xb_1, dxa_1, dxb_1, xa_2, xb_2, dxa_2, dxb_2)$ is ***well-defined*** if its height is positive, its widths along the x_1- and x_2- dimensions are positive, and the lengths of its bases along the x_1- and x_2- dimensions are nonnegative. We assume that the zoids we consider are well-defined. Define the ***projection trapezoid*** Z_1 of zoid Z along spatial dimension x_1 to be the 2D trapezoid that results from projecting Z onto the dimensions x_1 and t. The projection trapezoid Z_1 has two bases (sides parallel to the x_1 axis). We say that Z_1 is ***upright***, if the longer base of Z_1 corresponds to time ta, and ***inverted*** otherwise. These definitions can be extended to higher dimensional zoids. The ***space-time volume*** of a zoid is the product of its spatial volume and height.

Now suppose we assume ***no equivalence***, that is, no two zoids in the search domain have the same optimal plan. Then plan-finding under no equivalence does not need to store and look up plans of zoids. However, it can still incur a large memory overhead in maintaining the optimal plan, as stated in the following theorem.

Theorem 3
Consider a $(d+1)$-dimensional zoid Z with spatial volume n and height h. Let L be the average space-time volume of zoids that correspond to the leaves of an optimal plan for Z. Then the optimal plan has at least nh/L nodes.

Proof
The space-time volume of Z is given by nh. Since a leaf has space-time volume L on average, the optimal plan has nh/L leaves. □

More importantly, plan-finding under no equivalence (and without the divide-subsumption and favored-dimension properties) doesn't find plans for any benchmark within two days. Hence, Figures 7 and 8 do not report on the time measurements under no equivalence.

We analyze the memory and time overheads of space-time equivalence in the following.

Lemma 4
Consider a zoid Z with height h. The zoids in the search domain tree rooted at Z have $\Theta(\lg h)$ different heights.

Proof
The height of a zoid decreases only when it is cut in time. At each time cut, the TRAPPLE algorithm bisects a given zoid with height $h' > 1$ into two subzoids with heights $\lfloor h'/2 \rfloor$ and $\lceil h'/2 \rceil$. The result then follows from Lemma 1. □

Copyright © 0000 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe

Theorem 5
Consider a $(d+1)$-dimensional zoid Z with spatial volume n and height h. The search domain tree $\mathscr{T}$ rooted at Z has $O(2^d n \lg h)$ distinct zoids.

Proof
We count the number of distinct zoids for each possible height $h' \in \{h, \lfloor h/2 \rfloor, \lceil h/2 \rceil, \ldots, 1\}$. Let $w_i, 1 \leq i \leq d$ be the width of Z in spatial dimension x_i. Then, the number of upright (or inverted) distinct 2D projection trapezoids of a given height h' along the spatial dimension x_i is at most w_i, the width along that dimension. Hence the number of distinct $(d+1)$-dimensional zoids of height h' is at most $(2w_1 * 2w_2 * \cdots * 2w_d) = 2^d(w_1 * w_2 * \cdots * w_d) = 2^d n$. Since the zoids in $\mathscr{T}$ have $\Theta(\lg h)$ different heights from Lemma 4, the result follows. $\square$

Theorem 6
Consider a $(d+1)$-dimensional zoid Z with spatial volume n and height h. Procedure ZTUNE takes $O(2^d n^2 h)$ time to find an optimal plan for Z, under space-time equivalence.

Proof
To find the plan-finding time for Z, we sum the base-case costs of all distinct zoids in the search domain tree $\mathscr{T}$ rooted at Z. The base-case cost of a zoid is proportional to its space-time volume. We assume that the divide cost, link cost, and the costs of INSERT and LOOKUP procedures are negligible compared to the base-case cost of a given zoid. The base-case cost of a zoid of a given height h' is at most nh', the maximum space-time volume for height h'. From Theorem 5, there are at most $2^d n$ distinct zoids of height h'. Hence the sum of base-case costs of distinct zoids of height h' is at most $2^d n^2 h'$. And the sum of base-case costs of distinct zoids of all heights $h' \in \{h, \lfloor h/2 \rfloor, \lceil h/2 \rceil, \ldots, 1\}$ is at most $\sum_{h' \in \{h, \lfloor h/2 \rfloor, \lceil h/2 \rceil, \ldots, 1\}} 2^d n^2 h' = O(2^d n^2 h)$. $\square$

However, the memory and time bounds in Theorems 5 and 6 might not be tight. In specific, we conjecture that the memory overhead under space-time equivalence is $\Theta(n)$, which is also the memory overhead of TRAPPLE. We also considered ***time equivalent*** zoids, which are translations in just the time dimension, and a ***time equivalence*** property, which assumes that two time equivalent zoids have the same optimal plan. Time equivalence incurred significantly more memory and plan-finding time overheads, however, and failed to find a plan for any benchmark.

Handling Boundaries

STE, as defined, does not hold at the boundaries of the spatial grid. Recall that TRAP uses a faster base-case kernel function for zoids that lie in the interior of the grid, and a slower kernel function for zoids that impinge on the boundary. Consequently, two space-time equivalent zoids can have different base-case costs if one lies in the interior and the other impinges on the boundary. To address this anomaly, we use STE only for zoids that lie in the interior, and use a variant of STE called "boundary equivalence" for zoids that impinge on the boundary. Suppose we have two 3-dimensional zoids $Z \subseteq \mathbf{N} \times \mathbf{Z}^2$ and $Z' \subseteq \mathbf{N} \times \mathbf{Z}^2$, such that, Z and Z' impinge on the boundary in the x_1 dimension, but lie in the interior in the x_2 dimension. Z and Z' are ***boundary equivalent*** in x_1 if there exists a bijection $f : Z \to Z'$ and constants $T, X_2 \in \mathbf{Z}$ such that for all $(t, x_1, x_2) \in Z$, we have $f(t, x_1, x_2) = (t+T, x_1, x_2+X_2)$, that is, Z' can be a translation of Z in all dimensions except the spatial dimension x_1. The ***boundary equivalence*** property states that two zoids that are boundary equivalent in x_1 have the same optimal plan. These definitions can be adapted when the zoids impinge on the boundary in the x_2 dimension, or in both the x_1 and x_2 dimensions.

Discussion

Why does STE despite being a theoretical property hold in practice? STE assumes that the "context of execution" of two space-time equivalent zoids is the same, that is, the number of floating-point operations and memory accesses, the memory layout of the spatial grid, cache alignment of the grid points, and the states of the processor, memory hierarchy and other system components remain the same. The number of floating-point operations and memory accesses, which dominate the runtime of

Copyright © 0000 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe

```
DIVIDE-SUBSUMPTION(Z, z)
1  measurebase = TRUE
2  for z' ∈ z.children
3      if z'.choice ≠ −1 // test if a subzoid of Z was divided
4          measurebase = FALSE
5  if measurebase == TRUE
6      BASE-CASE (Z, z)
```

Figure 12: Pseudocode for divide-subsumption. DIVIDE-SUBSUMPTION takes as input a zoid Z and the root node z of a plan for Z.

stencil codes, and the memory layout of the spatial grid are the same for two space-time equivalent zoids. The cache alignment of the grid points, and the states of the processor, memory hierarchy and other system components might differ, during the execution of two space-time equivalent zoids. However, these effects do not dominate the runtime of stencil codes. Empirical evidence shown in Figure 11, which aggregates noise from all the pruning properties, indicates that STE holds reasonably well in practice.

6. DIVIDE-SUBSUMPTION PROPERTY

This section describes Ztune's "divide-subsumption" (DS) property, which speeds up plan-finding significantly, using the notion that it may not be necessary to find the base-case cost of every zoid in the search domain. For example, DS speeds up plan-finding for the APOP benchmark by a factor of at least 4900.

The ***divide-subsumption*** property states that executing the base case cannot be the optimal divide choice for a zoid Z, if it is not the optimal divide choice for a subzoid Z' of Z. The intuition is that procedure ZTUNE chose to divide Z', since Z' did not fit in cache and hence it was expensive to execute the base case on Z'. Since Z is larger than Z', Z will also not fit in cache and hence it will also be expensive to execute the base case on Z. Once ZTUNE chooses to divide a zoid Z' during the bottom-up traversal of a search domain, it can divide the ancestors of Z' in the search domain, and avoid finding their base-case costs under DS. The property cuts the search domain such that zoids that lie above the cut are divided, and zoids that lie below the cut are undivided and have the base case executed on them.

Figure 12 shows a DIVIDE-SUBSUMPTION procedure, which works as follows. Lines 2–4 perform the DS test. If a subzoid of Z was divided, then the procedure skips measuring the base-case cost of Z, by not calling BASE-CASE. Otherwise, line 6 calls BASE-CASE, which measures the base-case cost of Z. DIVIDE-SUBSUMPTION assumes that plans from all possible divide choices of Z except the base-case were already found, and that the current plan rooted at node z has the smallest cost of all such plans. To perform plan-finding using DS, line 21 of the ZTUNE procedure in Figure 5 can be replaced with a call to DIVIDE-SUBSUMPTION.

Figures 7 and 8 show the performance of plan-finding under DS and STE on the benchmarks. DS successfully finds plans for all benchmarks, and takes at most 15 hours to find a plan for a benchmark. It speeds up plan-finding under STE by a factor of 151–244. Significantly, faster plan-finding with DS does not affect the quality of the runtime.

Discussion

A few important design choices that we made in our implementation are described in the following. We do not assume DS for small zoids, since measuring their base-case costs can be error-prone. Under DS, plan-finding might erroneously choose to divide a zoid, whereas the correct choice would have been to execute the zoid as a base case. For example, a page fault during measurement can artificially increase the base-case cost of a zoid Z. DS would then divide Z and all the ancestors of Z in the choice domain, resulting in a sub-optimal plan. We say that two zoids Z_1, Z_2 are ***consecutive***

Copyright © 0000 John Wiley & Sons, Ltd.

Prepared using cpeauth.cls
DOI: 10.1002/cpe

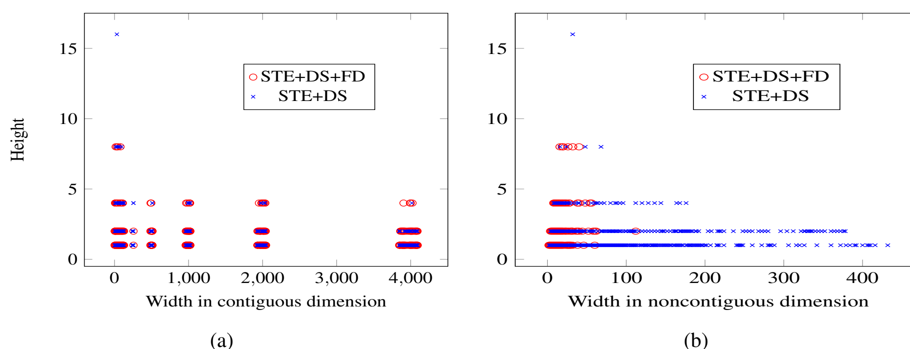

Figure 13: Comparison of base case sizes in the plans created with STE+DS and STE+DS+FD for the Heat4 benchmark with grid size 4096×4096, and 512 time steps, on Ivy Bridge. (a) Widths of base case zoids in the "contiguous" dimension. (b) Widths of base case zoids in the noncontiguous dimension.

descendants of a zoid Z_3, if Z_1 is a subzoid of Z_2 and Z_2 is a subzoid of Z_3. To make DS robust, we measure the base-case cost of each zoid in the choice domain, unless it has two consecutive descendants that were divided.

7. FAVORED-DIMENSION PROPERTY

This section describes an empirical property used by Ztune called "favored-dimension" (FD), which can greatly reduce plan-finding time, especially for stencils in higher dimensions. As an example, FD speeds up plan-finding for the Heat6 benchmark by a factor of 99. FD exploits architectural features of modern computers, such as vector units, cache blocking, and hardware prefetching, which accelerate performance when a processing core operates on consecutive memory locations. This section also reviews the empirical results that substantiate the effectiveness of the property.

Favored-dimension exploits the fact that when a processing core peforms computations on consecutive locations in the linear array of memory, the hardware operates considerably faster than when the computations are performed on nonconsecutive locations. Consequently, the way that the spatial grid is laid out in memory strongly influences performance. [20, 37] discuss this observation in tiling-based stencil codes. For a d-dimensional array, the ***stride*** of a dimension i is the distance in the linear array of memory between a grid point and an adjacent grid point, whose coordinates differ only in dimension i. We assume that a d-dimensional spatial grid is stored in memory as a linear array, where the dimensions are sorted by stride, with dimension 1 having the largest stride and dimension d — the ***contiguous*** dimension — having the smallest. For a 2-dimensional spatial grid, this layout corresponds to column-major order. These assumptions are without loss of generality, as the same optimizations could be implemented no matter how the dimensions are permuted. Figure 13 shows that an optimal plan found under STE and DS has base cases that are significantly longer in the contiguous dimension than in the noncontiguous dimension. Favored-dimension exploits this observation to prune the choice domain further.

The ***favored-dimension*** property for a $(d+1)$-dimensional zoid Z is illustrated in Figure 14 using procedure PRUNE-CHOICE, which works as follows. Line 2 defines S as the set of noncontiguous spatial dimensions along which Z can be divided. If the set S is not empty, then line 4 chooses a noncontiguous spatial dimension $i \in S$ arbitrarily. Line 5 returns a pruned set of divide choices, which includes the spatial dimension i and may include the time dimension 0 depending on whether time is a possible divide choice for Z. If the set S is empty, then possible divide choices for Z are either the contiguous dimension, or the time dimension, or both, or none. In this case, Line 7 returns the original set C of divide choices, without pruning. To perform plan-finding with favored-dimension, the call to CHOICE(Z) in line 7 of the ZTUNE procedure in Figure 5 can be replaced with a call to PRUNE-CHOICE(Z).

Copyright © 0000 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe

```
PRUNE-CHOICE(Z)
1  C = CHOICE(Z)
   // S is the set of noncontiguous dimensions in which Z can be divided
2  S = C ∩ {1, 2, ..., d−1}
3  if S ≠ ∅
4     Choose a dimension i ∈ S arbitrarily
5     return C ∩ {0, i}
6  else
7     return C
```

Figure 14: Pseudocode for favored-dimension. PRUNE-CHOICE takes as input a $(d+1)$-dimensional zoid Z and returns the set of divide choices for Z. Choice 0 corresponds to the time dimension, choices $1,2,\ldots,d-1$ correspond to the noncontiguous spatial dimensions, and choice d corresponds to the contiguous spatial dimension.

Favored-dimension does the following. It restricts that zoid Z be divided in at most 1 spatial dimension. To leverage the compiler and hardware optimizations in the contiguous dimension, the property avoids dividing Z along that dimension, where possible. Let P and P' be the minimum cost plans found with and without favored-dimension respectively, for zoid Z. Then favored-dimension assumes that the cost of plan P is no higher than the cost of plan P'.

Figures 7, 8, and 9 illustrate the performance of favored-dimension on the benchmarks. Favored-dimension speeds up plan-finding for 2 or higher dimensional problems by a factor of 1–99. Although procedure PRUNE-CHOICE allows us to pick a noncontiguous dimension arbitrarily in line 4 of Figure 14, for the experiments, we let the procedure choose a noncontiguous dimension $i \in S$, such that, i has the biggest stride among the dimensions in S. Choosing the noncontiguous dimension arbitrarily produces similar runtimes for the tuned TRAPPLE code.

8. EVALUATION

This section presents three different evaulations of the performance of Ztune. The first evaluation compares the performance of the pruned-exhaustive autotuning strategy in Ztune with that of the heuristic autotuning strategy in the OpenTuner [31] framework. The second evaluation compares the performance of the Ztuned divide-and-conquer TRAPPLE code with that of autotuned tiling-based stencil codes in Pluto [38, 39] and Patus [26]. Though making such a comparison is not the focus of this paper, we examined if the Ztuned divide-and-conquer TRAPPLE code is indeed competitive. The third evaluation reports on the performance of the Ztuned TRAPPLE code relative to Pochoir's default TRAP code on two more machines with different architectures.

Comparison with heuristic autotuning

In the first evaluation, we compare the pruned-exhaustive autotuning strategy in Ztune with the heuristic autotuning strategy in the OpenTuner framework. We briefly describe OpenTuner's search domain, which is different from the choice domain of Ztune. Whereas the introduction of the paper made an absolute comparison of the performance of Ztuned and OpenTuner tuned codes by running OpenTuner for 16 hours, this section makes a relative comparison of the tuned codes, by running OpenTuner for a few multiples of time longer than Ztune. We also report on the autotuning times taken by OpenTuner so that the OpenTuner tuned code achieves similar runtime as the Ztuned code on the benchmarks. Empirical evidence indicates that Ztune can autotune faster than OpenTuner, explore a more complex search domain, and generally produce tuned code that is faster.

Regrettably for scientific purposes, it is not feasible to configure OpenTuner to autotune TRAPPLE. To do so, one needs to parametrize the divide choice at each node in the choice domain, traversed by Ztune. Parametrizing the divide choice at each node creates a huge memory overhead,

Copyright © 0000 John Wiley & Sons, Ltd.
Prepared using cpeauth.cls

DOI: 10.1002/cpe

Benchmark	*1*	*2*	*4*	*8*	*16*	*32*
APOP	1.65	1.65	1.65	1.65	1.65	1.65
Heat1	2.37	2.37	1.61	1.61	1.06	1.00
Heat2	12.15	12.15	12.15	4.45	1.53	1.09
Life	3.97	3.15	1.29	1.23	1.00	1.00
Heat3	5.10	1.53	1.53	1.53	1.14	1.14
Heat4	1.67	1.67	1.67	1.67	1.67	1.35
Heat5	1.20	1.20	1.20	1.20	1.20	1.20
LBM	†4.35	†4.35	‡3.03	‡3.03	‡1.77	1.56
Wave	5.79	5.79	2.33	2.33	1.64	1.64
Heat6	1.94	1.93	1.73	1.25	1.14	1.14

Figure 15: Ratios of the runtimes of OpenTuner tuned and Ztuned codes on *Ivy Bridge*. The column headers indicate how many times longer OpenTuner tuned the benchmark than Ztune. The values indicate the ratios of runtimes of OpenTuner tuned TRAP code to the runtimes of Ztuned TRAPPLE code. A bigger ratio favors Ztune, and a smaller ratio OpenTuner. OpenTuner autotuned each benchmark using the IntegerValues, PowersOfTwo, and ListOfNumbers search domains, and the best OpenTuner tuned runtime among the three search domains was used to compute the ratio. Unannotated runtimes indicate PowersOfTwo search domain, and those annotated with the daggers † and ‡ indicate IntegerValues and ListOfNumbers search domains, respectively. Values are the best of two runs.

however, since the number of nodes in the choice domain is asymptotically larger than the space-time volume of the zoid Z at the root of the choice domain. Recall that TRAPPLE incurs memory overhead just proportional to the spatial volume of Z. Moreover, many of these parameters could not be effectively manipulated by OpenTuner, since their very existence might depend on decisions made higher in the recursion. For example, if a node executes a base case, the node has no descendants, and so the parameters that would be associated with the descendants do not affect the runtime. Nevertheless, the structure of OpenTuner provides no way for OpenTuner *not* to keep twiddling these parameters and wasting time, even though they are irrelevant to the optimal plan.

Consequently, we had to restrict OpenTuner's heuristic search to the domain of base-case sizes, and optimally coarsen the base case of recursion in the TRAP algorithm. Two OpenTuner parameters were created for each dimension of the space-time grid, one for the base-case size of zoids that impinge on the boundary, and the other for the base-case size of zoids that do not impinge on the boundary. This yielded a sizable search domain for OpenTuner, albeit smaller than Ztune's. OpenTuner uses an ensemble of search techniques that include greedy mutation, differential evolution, and hill-climbing methods.

To provide a fair comparison with Ztune, we configured OpenTuner to tune the TRAP algorithm using three search domains. Two of the search domains are relatively large in size, while the third one is small. We contend that this configuration is fair to OpenTuner, since smaller search domains allow OpenTuner to tune faster, while the larger search domains allow it to find a solution that is closer to optimal at the expense of higher tuning times. The largest search domain defined in OpenTuner is called ***IntegerValues***. The parameter lower and upper bounds were set to cover the grid size in each dimension. OpenTuner can choose any integer value within these bounds. Given a $(d+1)$-dimensional zoid Z with height h, and whose spatial widths are proportional to h, the search domain of IntegerValues has $O(h^d h^d h) = O(h^{2d+1})$ different base-case sizes, where the factor 2 is due to two parameters being tuned in each spatial dimension. In contrast, Ztune's choice domain is larger with $\Omega(2^{h/2})$ different plans. Many values within the lower and upper bounds can be redundant, however, since TRAP divides the spatial and temporal dimensions in a fixed fashion. The next smaller search domain called ***ListOfNumbers*** uses the actual widths and heights of zoids traversed by Ztune as the possible values for the parameters along the corresponding dimensions. This reduces the size of the search domain for Z to $O(h^{2d} \lg h)$, since there are only $\Theta(\lg h)$ different heights in the choice domain as shown in Lemma 4. The smallest search domain is called ***PowersOfTwo***. Similar to the IntegerValues search domain, the parameter lower and upper bounds in PowersOfTwo were set to cover the grid size in each dimension, but only power-of-2

Copyright © 0000 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe

Benchmark	*Ztune (minutes)*	*OpenTuner (hours)*
APOP	0.58	37.93
Heat1	0.33	0.15
Heat2	6.58	4.17
Life	4.40	1.32
Heat3	1.09	2.66
Heat4	0.79	>100
Heat5	6.55	48.75
LBM	0.56	0.65
Wave	51.90	>100
Heat6	5.75	>100

Figure 16: Tuning time comparison of OpenTuner and Ztune on *Ivy Bridge*. The second column shows the Ztune tuning times in minutes. The third column shows the number of hours OpenTuner needs to autotune, so that the OpenTuner tuned code achieves the same runtime as the Ztuned code. OpenTuner data was obtained from tuning each benchmark once. Due to the time-consuming nature of the experiment, we couldn't run OpenTuner more.

	Tuned runtimes			*Tuning quality*	
Benchmark	*Ztune*	*Pluto*	*Patus*	*Ztune*	*Pluto*
Heat1	1.24	0.93	6.49	0.93	0.88
Heat2	35.91	39.18	—	0.86	0.88
Heat3	17.23	44.10	—	0.87	0.87
Heat4	9.99	23.51	—	0.65	0.77

Figure 17: Comparison of the tuned runtimes (in seconds) of stencil codes under different autotuners, and "tuning quality" of the autotuners on 4 Heat benchmarks, on the Haswell machine. The headers *Ztune*, *Pluto*, and *Patus* under *Tuned runtimes* indicate the runtimes of the stencil codes autotuned with Ztune, Pluto's built-in autotuner, and Patus respectively. A dash sign (—) indicates that the benchmark could not be specified in Patus since it is periodic, and hence the runtime is unknown. The reported numbers are the better of two runs. The header *Ztune* under *Tuning quality* indicates the ratio of the runtime of the Ztuned TRAPPLE code to the runtime of Pochoir's TRAP code. Similarly, the header *Pluto* under *Tuning quality* indicates the ratio of the runtime of Pluto's autotuned tiled code to the runtime of Pluto's default tiled code. A lower ratio indicates that the autotuned code runs faster.

integer values can be chosen between the bounds. This reduces the size of the search domain for Z further to $O(\lg^{2d+1} h)$.

Whereas Ztune runs its natural time to autotune, OpenTuner can be run for any length of time. Figure 15 shows the ratios of the runtimes of OpenTuner tuned TRAP code to the runtimes of Ztuned TRAPPLE code, where OpenTuner was run for $1, 2, \ldots, 32$ times longer than Ztune. In general, the longer OpenTuner tunes, faster are its tuned codes. As can be seen from Figure 15, despite an enormous advantage in tuning time, an OpenTuner tuned code doesn't beat the Ztuned code. We also infer that the PowersOfTwo configuration for OpenTuner performs the best for most of the benchmarks.

Figure 16 shows how long OpenTuner must run to produce code that runs as fast as the Ztuned code. For example, to obtain the same runtime as the Ztuned code, OpenTuner must tune APOP for almost 38 hours, whereas Ztune ran in under a minute. To achieve comparable runtimes to the Ztuned code, OpenTuner has to tune most of the benchmarks for over an hour. Despite tuning for 100 hours, the OpenTuner tuned Heat4, Wave, and Heat6 benchmarks couldn't achieve the performance of the Ztuned code. We conclude that the pruned-exhaustive tuning strategy in Ztune can tune divide-and-conquer stencil problems considerably faster than the heuristic tuning strategies in OpenTuner.

Copyright © 0000 John Wiley & Sons, Ltd.
Prepared using cpeauth.cls

DOI: 10.1002/cpe

Benchmark	*Haswell*	*Opteron*	*Benchmark*	*Haswell*	*Opteron*
APOP	0.95	0.94	Heat4	0.68	0.71
Heat1	0.94	0.93	Heat5	0.86	0.90
Heat2	0.88	0.88	LBM	0.98	0.99
Life	1.00	1.01	Wave	0.85	0.98
Heat3	0.87	0.96	Heat6	0.90	0.93

Figure 18: Performance of the Ztuned TRAPPLE code relative to Pochoir's default hand-tuned TRAP code on the Haswell and Opteron machines. The headers *Haswell* and *Opteron* indicate the ratio of the runtime of Ztuned TRAPPLE code to the runtime of Pochoir's TRAP code on those machines respectively. A lower ratio indicates that the Ztuned code runs faster than Pochoir's code. The geometric mean of the ratios on Haswell is 0.89, and that on Opteron is 0.92. The reported ratios are the better of two runs.

Comparison with autotuned tiling-based stencil codes

The second evaluation compares the performance of Ztuned divide-and-conquer TRAPPLE code with that of autotuned tiling-based stencil codes. Though making such a comparison is not the focus of this paper, we examined if the Ztuned TRAPPLE code is indeed competitive. Figure 17 compares the runtime of Ztuned TRAPPLE code with the runtimes of autotuned tiling-based codes in Pluto [38, 39] and Patus [26]. We ran the comparison on a few Heat benchmarks, which could be easily specified in Pluto and Patus. Since Pluto recommends using a more recent version of Intel compiler, we used ICC 15.0.6 to compile all the 3 autotuners, and used the same compiler flags. Pluto's built-in autotuner searches a fixed set of tile sizes to find an optimal tile size. Figure 17 reports Pluto's best runtime from among different settings like "tiled" and "lbpar". Since Patus doesn't support periodic boundary conditions, it couldn't be used to autotune the periodic benchmarks Heat2, Heat3, and Heat4. The slow runtime of the Patus tuned Heat1 benchmark is probably due to the fact that it doesn't autotune for temporal blocking. Figure 17 also compares the "tuning quality", which is the ratio of the runtime of the autotuned code to the runtime of the default code, of Ztune and Pluto. Since Patus doesn't have a default code to run, we couldn't report on its tuning quality. Figure 17 indicates that divide-and-conquer based TRAPPLE codes and Ztune are competitive with their tiling based counterparts for the benchmarks considered. It would be premature to conclude that one strategy is faster than the other, however, since this comparison is not exhaustive and not the focus of this paper. Previous work [37, 40] describes experiments where tiling is faster than divide-and-conquer strategies. We thank Uday Bondhugula for graciously clarifying all our questions about autotuning the benchmarks in Pluto, and Matthias Christen for helping with Patus related queries.

Performance of Ztune on other architectures

The third evaluation compares the performance of the Ztuned TRAPPLE code with that of Pochoir's TRAP code on two more machines — Haswell and Opteron, whose specifications are shown in Figure 19. Recall that the introduction of the paper reported on the performance of Ztune on the Nehalem and Ivy Bridge machines. Figure 18 shows the performance of Ztune on the Haswell and Opteron machines. The Ztuned code is on average 11% faster on Haswell and 8% faster on Opteron than Pochoir's code.

9. CONCLUDING REMARKS

We have presented Ztune, a pruned-exhaustive autotuner for serial divide-and-conquer stencil computations, and described three properties namely space-time equivalence, divide subsumption, and favored dimension, which improve the performance of Ztune significantly. Pruned-exhaustive autotuning in Ztune exploits its knowledge of divide-and-conquer stencil codes, to autotune faster and produce tuned codes with similar or better runtimes than heuristic autotuning. Heuristic autotuners are nevertheless useful tools to tune a broad range of applications, where domain-specific autotuning tools might not necessarily exist.

Copyright © 0000 John Wiley & Sons, Ltd.
Prepared using cpeauth.cls

	Nehalem	*Ivy Bridge*	*Haswell*	*Opteron*
Manufacturer	Intel	Intel	Intel	AMD
CPU	Xeon X5650	Xeon E5-2695 v2	Xeon E5-2666 v3	Opteron 6376
Clock	2.66 GHz	2.4 GHz	2.90 GHz	2.3 GHz
Hyperthreading	Disabled	Enabled	Enabled	Enabled
Turbo Boost	Disabled	Disabled	Disabled	Enabled
Processor cores	12	24	18	32
Sockets	2	2	2	4
L1 data cache/core	32 KB	32 KB	32 KB	16 KB
L2 cache/core	256 KB	256 KB	256 KB	2 MB
L3 cache/socket	12 MB	30 MB	25 MB	6 MB
DRAM	48 GB DDR3	128 GB DDR3	58 GB DDR3	256 GB DDR3
Compiler	ICC 13.1.1	ICC 13.1.1	ICC 13.1.1	ICC 13.1.1
Operating system kernel	Linux 3.13.0	Linux 3.13.0	Linux 4.1.10	Linux 2.6.32
Advanced Vector Extensions	No	Yes	Yes	Yes

Figure 19: Specifications of the machines used for benchmarking. We disabled Turbo Boost, where possible, to enhance the reliability of time measurements.

More generally, we have presented a theoretical autotuning framework that can be used for tuning many divide-and-conquer codes in scientific computing like matrix multiplication, convolution, and dynamic-programming problems. We have extended Ztune to autotune divide-and-conquer matrix-vector product and matrix multiplication, the preliminary results of which can be found in [41]. Augmenting Ztune to autotune parallel divide-and-conquer codes is an interesting research area. Parallel codes introduce a host of issues like memory bandwidth saturation, communication overhead, and work-span optimization.

The tuning strategy in Ztune has some obvious drawbacks. Ztune doesn't extrapolate its tuning results, but autotunes every problem from scratch. Though it autotunes faster, the autotuning time can be reduced significantly by extrapolation.

REFERENCES

1. Bleck R, Rooth C, Hu D, Smith LT. Salinity-driven thermocline transients in a wind- and thermohaline-forced isopycnic coordinate model of the North Atlantic. *J. of Phys. Oceanography* 1992; **22**(12):1486–1505.
2. Datta K, Murphy M, Volkov V, Williams S, Carter J, Oliker L, Patterson D, Shalf J, Yelick K. Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures. *SC*, ACM/IEEE, 2008; 4:1–4:12.
3. Dursun H, Nomura Ki, Peng L, Seymour R, Wang W, Kalia RK, Nakano A, Vashishta P. A multilevel parallelization framework for high-order stencil computations. *Euro-Par*, 2009; 642–653.
4. Dursun H, Nomura Ki, Wang W, Kunaseth M, Peng L, Seymour R, Kalia RK, Nakano A, Vashishta P. In-core optimization of high-order stencil computations. *PDPTA*, 2009; 533–538.
5. Frigo M, Strumpen V. Cache oblivious stencil computations. *ICS*, ACM, 2005; 361–366.
6. Frigo M, Strumpen V. The cache complexity of multithreaded cache oblivious algorithms. *Theory of Computing Systems* 2009; **45**(2):203–233.
7. Kamil S, Datta K, Williams S, Oliker L, Shalf J, Yelick K. Implicit and explicit optimizations for stencil computations. *MSPC*, ACM, 2006; 51–60, doi:http://doi.acm.org/10.1145/1178597.1178605.
8. Kamil S, Husbands P, Oliker L, Shalf J, Yelick K. Impact of modern memory subsystems on cache optimizations for stencil computations. *MSP*, ACM, 2005; 36–43, doi:http://doi.acm.org/10.1145/1111583.1111589.
9. Kamil S, Chan C, Oliker L, Shalf J, Williams S. An auto-tuning framework for parallel multicore stencil computations. *IPDPS*, IEEE, 2010; 1–12.
10. Krishnamoorthy S, Baskaran M, Bondhugula U, Ramanujam J, Rountev A, Sadayappan P. Effective automatic parallelization of stencil computations. *PLDI*, ACM, 2007.
11. Nakano A, Kalia RK, Vashishta P. Multiresolution molecular dynamics algorithm for realistic materials modeling on parallel computers. *Comp. Phys. Comm.* 1994; **83**(2-3):197–214.
12. Nitsure A. Implementation and optimization of a cache oblivious lattice Boltzmann algorithm. Master's Thesis, Institut für Informatic, Friedrich-Alexander-Universität Erlangen-Nürnberg 2006.
13. Peng L, Seymour R, Nomura Ki, Kalia RK, Nakano A, Vashishta P, Loddoch A, Netzband M, Volz WR, Wong CC. High-order stencil computations on multicore clusters. *IPDPS*, IEEE, 2009; 1–11.
14. Taflove A, Hagness SC. *Computational Electrodynamics: The Finite-Difference Time-Domain Method*. Artech, 2000.
15. Tang Y, Chowdhury RA, Kuszmaul BC, Luk CK, Leiserson CE. The Pochoir stencil compiler. *SPAA*, ACM, 2011; 117–128.
16. Williams S, Carter J, Oliker L, Shalf J, Yellick K. Optimization of a lattice Boltzmann computation on state-of-the-art multicore platforms. *JPDC* 2009; **69**(9):762–777.
17. Malas T, Hager G, Ltaief H, Stengel H, Wellein G, Keyes D. Multicore-optimized wavefront diamond blocking for optimizing stencil updates. *SIAM Journal on Scientific Computing* 2015; **37**(4):439–464.

Copyright © 0000 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe

18. Malas TM, Hornich J, Hager G, Ltaief H, Pflaum C, Keyes DE. Optimization of an electromagnetics code with multicore wavefront diamond blocking and multi-dimensional intra-tile parallelization. *arXiv:1510.05218* 2015; URL `http://arxiv.org/abs/1510.05218`.
19. Song Y, Li Z. New tiling techniques to improve cache temporal locality. *PLDI*, ACM, 1999; 215–228.
20. Rivera G, Tseng C. Tiling optimizations for 3D scientific computations. *SC*, ACM/IEEE, 2000; 32:1–32:23.
21. Frigo M, Leiserson CE, Prokop H, Ramachandran S. Cache-oblivious algorithms. *FOCS*, IEEE, 1999; 285–297.
22. Frigo M. A fast Fourier transform compiler. *ACM SIGPLAN Notices* May 1999; **34**(5):169–180.
23. Frigo M, Johnson S. The design and implementation of FFTW3. *Proceedings of the IEEE* 2005; **93**(2):216–231.
24. Whaley RC, Dongarra J. Automatically tuned linear algebra software. *SC*, ACM, 1998; 1–27.
25. Vuduc R, Demmel JW, Yelick KA. OSKI: A library of automatically tuned sparse matrix kernels. *J. of Phys.*, vol. 16, 2005; 521.
26. Christen M, Schenk O, Burkhart H. Patus: A code generation and autotuning framework for parallel iterative stencil computations on modern microarchitectures. *IPDPS*, IEEE, 2011; 676–687.
27. Kamil SA. Productive high performance parallel programming with auto-tuned domain-specific embedded languages. PhD Thesis, University of California, Berkeley 2012.
28. Moura JMF, Singer B, Xiong J, Johnson J, Padua D, Veloso M, Johnson RW. SPIRAL: A generator for platform-adapted libraries of signal processing algorithms. *Int. J. of High Perf. Comp. Appl.* 2004; **18**(1):21–45.
29. Ansel J, Chan C. PetaBricks: Building adaptable and more efficient programs for the multi-core era. *XRDS* 2010; **17**(1).
30. Ţăpuş C, Chung IH, Hollingsworth JK. Active Harmony: Towards automated performance tuning. *SC*, ACM/IEEE, 2002; 1–11.
31. Ansel J, Kamil S, Veeramachaneni K, O'Reilly UM, Amarasinghe S. OpenTuner: An extensible framework for program autotuning. *Technical Report TR-2013-026*, MIT CSAIL 2013.
32. Epperson JF. *An Introduction to Numerical Methods and Analysis*. Wiley-Interscience, 2007.
33. John C. *Options, Futures, and Other Derivatives*. Prentice Hall, 2006.
34. Gardner M. Mathematical Games. *Scientific American* 1970; **223**(4):120–123.
35. Mei R, Shyy W, Yu D, Luo L. Lattice Boltzmann method for 3-D flows with curved boundary. *J. of Comput. Phys* 2000; **161**(2):680–699.
36. Micikevicius P. 3D finite difference computation on GPUs using CUDA. *GPGPU*, ACM, 2009; 79–84.
37. Datta K, Kamil S, Williams S, Oliker L, Shalf J, Yelick K. Optimization and performance modeling of stencil computations on modern microprocessors. *SIAM Rev.* 2009; **51**(1):129–159.
38. Bondhugula U, Baskaran M, Krishnamoorthy S, Ramanujam J, Rountev A, PSadayappan. Automatic transformations for communication-minimized parallelization and locality optimization in the polyhedral model. *International Conference on Compiler Construction (ETAPS CC)*, ACM, 2008; 132–146.
39. Bondhugula U, Hartono A, Ramanujam J, Sadayappan P. A practical automatic polyhedral parallelizer and locality optimizer. *PLDI*, ACM, 2008; 101–113.
40. Bondhugula U, Bandishti V, Cohen A, Potron G, Vasilache N. Tiling and optimizing time-iterated computations on periodic domains. *PACT*, ACM, 2014; 39–50.
41. Pantawongdecha P. Autotuning divide-and-conquer matrix-vector multiplication. Master's Thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology Jun 2016.

Copyright © 0000 John Wiley & Sons, Ltd.
Prepared using cpeauth.cls

DOI: 10.1002/cpe