Cetus: A Source-to-Source Compiler Infrastructure for Multicores

Hansang Bae, Leonardo Bachega, Chirag Dave, Sang-Ik Lee, Seyong Lee,
Seung-Jai Min, Rudolf Eigenmann and Samuel Midkiff

Purdue University*

Abstract. We describe the Cetus compiler infrastructure and its use in a number of transformation tasks for multicore architectures. The original intent of Cetus was to serve as a parallelizing compiler. In addition, the infrastructure has been used to build translators for programs written in the OpenMP directive language to be compiled onto multicore architectures. They include a direct OpenMP translator for current multicores, an OpenMP to MPI translator for many-cores exhibiting disjoint address spaces, and a translator for OpenMP onto GPU architectures. We are also building autotuning capabilities into Cetus, which can defer compile-time optimization decisions to runtime. This feature is especially important for heterogeneous multicore architectures. We will describe the organization of the Cetus infrastructure and present preliminary results of several application projects.

1 Introduction

Cetus is a source-to-source restructuring compiler infrastructure for C programs, and is a follow-on project to the Polaris Fortran translator [1, 2]. The driving motivation was the need for a source-level compiler infrastructure that facilitates advanced optimizations for parallel programs written in C. Cetus has already been used for a number of applications, several of which we will describe.

Cetus had originally been created in a Purdue advanced compiler class project and has evolved into a fairly robust infrastructure. At first, the manpower behind the Cetus development was all "volunteer work" by several dedicated graduate students. Recently, the project has obtained funding from the U.S. National Science Foundation to become a community resource. This grant allows us to improve the robustness of Cetus, respond to user requests, and add new features.

Cetus is already in use by a number of research groups in the U.S. and worldwide [3–6]. Increasing the user community is one goal of this paper. To this end, this paper describes the Cetus Community Portal in Section 2, the Cetus internal organization in Section 3, current analysis and transformation capabilities in Sections 4 and 5, respectively, and applications projects in Section 6.

* This work was supported, in part, by the National Science Foundation under grants No. 0429535-CCF, 0650016-CNS, CNS-0751153, and CNS-0707931.

Cetus is being used and extended in ways beyond those described in this paper. Notably, in ongoing work we are adding passes for improved alias and data-dependence analysis as well as additional parallelizing transformations. Other projects extend Cetus to related languages, such as C++ and Java, and dialects, such as C for GPUs.

2 Cetus Community Portal

Developing a dependable community support system and reaching out to Cetus users is one of our primary goals. As we continue to build more functionality, it is crucial to be able to cater to the needs of our users and influence our development with their feedback. The Cetus Community Portal serves this purpose [7]. The infrastructure is available for download at this portal and is released under the Artistic License, details of which can be found on the website.

2.1 Documentation

The Cetus website provides documentation that assists with installing and running the compiler. The Compiler manual provides sections on the Cetus architecture as well as important information to help pass writers with incorporating new analysis and transformation passes using the existing Cetus API. The Cetus API is made available in Javadoc format via the website.

2.2 Community Support

We have now incorporated a strong user feedback system within the Cetus portal to additionally help us with answering questions from users and dealing with the issues they encounter as they continue to use our framework to meet their needs.

Bugzilla is a utility that allows users to submit bug reports. Users can also inform the Cetus team about issues they have come across while using the compiler, to see the progress related to submitted bugs and to receive feedback when the bugs have been fixed. This allows us to interact with our customer base while developing the infrastructure with the right needs in mind and to involve users in the development process of Cetus.

Cetus-users Mailing List can now be used to discuss ideas, new functionality and research related to Cetus that would benefit the entire research community as well as the users of Cetus. This can be considered as a robust means of information exchange that would help incorporate new ideas into Cetus.

3 Cetus Organization and Internal Representation

3.1 Cetus Class Hierarchy

Cetus' internal program representation (IR) is implemented in the form of a Java class hierarchy. There is complete data abstraction, with pass writers only

manipulating the IR through access functions. An early version of this class hierarchy was described in [8]. We briefly mention a few changes with respect to this version.

- *Symbol table:* Cetus' symbol table functionality provides information about identifiers and data types. Its implementation makes direct use of the information stored in declaration statements stored in the IR. There is no separate and redundant symbol table storage.
- *Traversable objects:* All Cetus IR objects are derived from a base class "Traversable". This class provides the functionality to iterate over lists of objects in a generic way.
- *Annotations:* Comments, pragmas, directives, and other types of auxiliary information about IR objects can be stored in *annotation objects*. They take the form of declarations. An annotation may be associated with a statement (e.g., info about an OpenMP directive belonging to a `for` statement) or may stand independently (e.g., a comment line).
- *Printing:* The printing functions have been extended to allow for flexible rendering of the IR classes.

3.2 Symbolic Manipulation

Like its predecessor Polaris [1, 2], a key feature of Cetus is its ability to reason about the represented program in symbolic terms. For example, program analysis and optimization techniques at the source level often require the expressions in the program to be in a simplified form. A specific example is a data dependence test which collects the coefficients of affine subscript expressions to interface with the underlying data dependence solvers. Cetus supports such expression manipulations by providing a set of utilities that simplify and normalize symbolic expressions. The examples in Figure 1 show the capabilities of the simplification utility. While most Cetus passes make use of these capabilities, they are key for the Symbolic Range Analysis Pass, described in Section 4.3.

`1+2*a+4-a`	⇒	`5+a`	(folding)
`a*(b+c)`	⇒	`a*b+a*c`	(distribution)
`(a*2)/(8*c)`	⇒	`a/(4*c)`	(division)
`(1-a)<(b+2)`	⇒	`(1+a+b)>0`	(normalization)
`a && 0 && b`	⇒	`0`	(short-circuit evaluation)

Fig. 1. Symbolic simplification/normalization examples.

4 Program Analysis Capabilities

Advanced program analysis capabilities are essential to Cetus; they will grow through our ongoing efforts and by incorporating passes developed by the Cetus user community. Here we are describing some basic analyses, including a

data flow framework, array section analysis, symbolic range analysis, and data dependence analysis.

4.1 Array Section Analysis

Array sections describe the set of array elements that are accessed by program statements. Array section analysis enables the Cetus passes to deal with sub-arrays. This capability increases the accuracy of analysis passes and enhances the efficiency of the transformation passes, compared to a name-based array analysis.

Cetus' array section analysis pass performs a conservative `use/def` analysis of array variables for a given *Traversable* input program. Before array section analysis is applied to a loop, all array subscript expressions are simplified. Array section analysis then collects and merges the obtained range information of array accesses to find the `may-use` and `may-def` set of array variables, where array variables are expressed in terms of array sections. Figure 2 shows the result of array section analysis for an example loop.

```
c = 2;
N = 100;
#pragma cetus USE(A[0:100][0:100]) DEF(B[1:99][1:99])
for (i=1; i<N; i++) {
  for (j=1; j<N; j++) {
    B[c*i - i][j] = (A[i-1][j] + A[i+1][j] + A[i][j-1] + A[i][j+1])/4;
  }
}
```

Fig. 2. Array section analysis example

4.2 Data Dependence Analysis

The data dependence analysis framework gathers dependence information for array accesses within loop nests and creates a data dependence graph. The framework comprises of a *data dependence test driver* that acts as a wrapper around conventional array subscript dependence tests, such as the Banerjee test and the GCD test.

Building the framework: The driver iterates over all loops in the program IR and identifies eligible loops. Eligibility currently defines the scope of dependence testing in Cetus. For example, we handle perfect loop nests and loops in the form `for(i=lb;i<ub;i++)` (called as canonical loops). Loop information (such as loop bounds, loop step and enclosing loops) and array access-related information (such as array references, enclosing loops and parent statements) is collected

in data structures and provided as input to the dependence test interface described below. Also, in order to accurately identify dependences, the framework takes into account, information provided by the privatization, reduction and alias analyses.

Execution of the dependence tests: The tests try to disprove dependence between a pair of array accesses and, if unable to do so, return a dependence vector representing the direction of dependence in each dimension of the iteration space spanned by the enclosing loop nest. The current implementation is based on the general algorithm described in Section 3.6 of [9]. The algorithm encompasses the testing of array accesses with multiple subscripts, each of these being an affine expression of the indices in the iteration space. The subscripts are first split into *partitions* (subsets of subscripts that can be tested independently), followed by a sequence of independent subscript tests for each partition. The result of each subscript test is a dependence vector associated with the subscript pair. At the end, the dependence vectors from all independent subscript tests are merged together into a single set of dependence vectors. Currently, Cetus includes the the Banerjee test as described in [9, 10] and our data dependence framework permits easy expansion to other subscript tests, such as the GCD test, the Omega test [11] and the Range test [12].

Dependence Graph Interface: The output of testing is an edge-based *Dependence Graph* (DG) [10] for dependent *array references* within eligible loop nests, i.e. we have a loop-based data dependence graph for the entire program. The DG is provided with interface functions that enable pass writers to extract dependence related information required for subsequent analyses and transformation. For example, statement-related dependence information can be extracted from the array-based dependence edges for transformations such as loop fusion or loop distribution. The graph interface is under development, and currently supports interface routines that are used by the Cetus loop parallelizer.

4.3 Range Analysis

The goal of Range Analysis is to collect, at each program statement, a map from integer-typed scalar variables to their symbolic value ranges, represented by a symbolic lower bound and an upper bound. In other words, a symbolic value range expresses the relationship between the variables that appear in the range. We use a similar approach described in Symbolic Range Propagation [13], with necessary adjustment for the C language, to compute the set of value ranges at each statement. The set of value ranges at each statement can be used in several ways. Pass writers can directly query the symbolic bound of a variable or can compare two symbolic expressions using the constraints given by the set of value ranges. For instance, the comparison algorithm can conclude that $v1 + v2 \geq v3$ with a supporting set of value ranges $\{v1 = [0, 10], v2 = v1 - 5, v3 = -5\}$ since the value range of the expression $v1 + v2$ is $[-5, 5]$.
The high-level algorithm of the range analysis performs fix-point iteration in two

phases when propagating the value ranges throughout the program. The first phase applies widening operations at nodes that have incoming back edges to guarantee the termination of the algorithm while the second phase compensates the loss of information due to the widening operations by applying narrowing operation to the node on which widening has occurred [14]. During the fix-point iteration, the value ranges are merged at nodes that have multiple predecessors, and outgoing value ranges are computed by symbolically executing the statement. Two typical types of program semantics that cause such changes of value ranges are constraints from conditional expressions and assignments to variables.

5 Basic Parallelizing Transformation Passes

We briefly describe the algorithms used in the current Cetus implementation for the basic three parallelizing transformation techniques: privatization, reduction variable recognition, and induction variable substitution. These are the techniques found most important in automatic parallelizing compilers [15]. In ongoing work we are developing techniques that can enhance these transformations further, including including inter-procedural analysis and advanced alias analysis.

5.1 Privatization

The high-level algorithm in Figure 3 describes the process of detecting privatizable variables, both scalars and array sections, in a loop. The set operations that appear in the algorithm are performed on the array sections if the variable is an array. We use the power of symbolic analysis techniques in Cetus to make the symbolic section operation possible. For example, $[1 : m] \cap [1 : n]$ results in $[1 : n]$ if the expression comparison tool with the supporting symbolic range environment can decide $n \leq m$.

The algorithm traverses a loop nest from the innermost to the outermost loop. At each level, it first collects *definitions* (write references) and *uses* (read references) in the loop body. Uses that are covered by prior definitions in the control flow qualify a variable as private for the current loop. The other uses are *upward exposed*, creating read references seen in the outer loop. Then, the algorithm selects a set of candidate variables by collecting defined scalars, defined arrays with loop-invariant sections, and private variables in the inner loops (`CollectCandidates`). The private variables are computed by eliminating upward exposed uses from the candidate variables, and the private variables that are used after the loop are marked as last-private. Last, the algorithm aggregates all these array sections over the loop iteration space, creating the array sections that are written and upward exposed for the entire loop. The aggregation for the written sections (*DEF*) computes the must-defined regular sections of the arrays over the entire loop iteration while the aggregation of upward-exposed sections (*UEU*) requires only the conservative value ranges of the sections (may-used sections). Notice that the *UEU* set of an inner loop becomes the *USE* set of

```
procedure Privatization(L, LIVE)
  input : L, LIVE
  output : DEF[L], UEU[L], PRI[L]
  side effect : L is annotated with PRI[L] and LPRI[L]

  // L: Loop, LIVE: Live-out variables, EXIT: Loop exits, PRED: Predecessors
  // KILL: Killed set due to modified variables in the section representation
  // DEF: Defined set, USE: Used set, UEU: Upward-exposed uses
  // PRI: Private variables, LPRI: Live-out private variables

  // 1. Privatize inner loops.
  foreach direct inner loop l in L
    (DEF[l], USE[l], PRI[l]) = Privatization(l, LIVE)

  // 2. Create a CFG of the loop body with collapsed inner loops.
  G(N, E) = BuildCFG(L)

  // 3. Compute must-defined set DEF prior to each node.
  Iteratively solve data flow equation DEF for node n ∈ N.
    DEF_in[n] = ⋂_{p∈PRED[n]} DEF_out[p]
    DEF_out[n] = (DEF_in[n] − KILL[n]) ∪ DEF[n]

  // 4. Compute DEF[L], UEU[L], PRI[L].
  DEF[L] = ⋂_{n∈EXIT[L]} DEF_out[n]
  UEU[L] = ⋃_{n∈N} (USE[n] − DEF_in[n])
  PRI[L] = CollectCandidates(L, DEF, PRI)
  PRI[L] = PRI[L] − UEU[L]
  LPRI[L] = PRI[L] ∩ LIVE[L]
  PRI[L] = PRI[L] − LPRI[L]
  AnnotatePrivate(L, PRI[L], LPRI[L])

  // 5. Aggregate DEF and USE over the iteration space
  DEF[L] = AggregateDEF(DEF[L])
  UEU[L] = AggregateUSE(UEU[L])
  return (DEF[L], UEU[L], PRI[L])
end procedure
```

Fig. 3. Privatization Algorithm. This is a simplified version of the original algorithm in [16].

the collapsed node while the algorithm traverses the outer loop. This algorithm is a slightly simpler version of the one described in [16].

5.2 Reduction Variable Recognition

We implemented the algorithm described in [17]. Essentially, a variable `sum` must satisfy two criteria to be an additive reduction variable.

```
// lhse/rhse extracts the left/right-hand side expression of an assignment expr.
//      (note, assignments are referred to as expressions rather than statements)
// findREF(X) returns the set of USE/DEF references in a given expression X.
// "−" is a symbolic subtraction operator.
// getSymbol(X) returns the base symbol of expression X (a scalar or an array)

procedure RecognizeReductions(L)
  Input : Loop L
  Side-effect : adds reduction annotations for L in the IR

  REDUCTION = {} // set of candidate reduction expressions
  REF = {} // set of non-reduction variables referenced in L
  foreach expr in L
    localREFs = findREF(expr)
    if (expr is AssignmentExpression)
      candidate = lhse(expr)
      increment = rhse(expr) − candidate
      if ( !(getSymbol(candidate) in findREF(increment)) )
        // criterion1 is satisfied
        REDUCTION = REDUCTION ∪ candidate
        localREFs = findREF(increment)
    REF = REF ∪ localREFs // collect non-reduction references for criterion 2

  foreach expr in REDUCTION
    if ( ! (getSymbol(expr) in REF) )
      // criterion 2 is satisfied
      if (expr is ArrayAccess AND expr.subscript is loop-variant)
        CreateAnnotation(sum-reduction, ARRAY, expr)
      else
        CreateAnnotation(sum-reduction, SCALAR, expr)
end procedure
```

Fig. 4. Algorithm to recognize additive scalar and array reduction variables in a loop.

- criterion 1: the loop contains one or several assignment expressions of the form `sum=sum+`*increment*, where *increment* is typically a real-valued, loop-variant expression.
- criterion 2: `sum` does not appear anywhere else in the loop.

The reduction variable `sum` can be a scalar or an array expression. If `sum` is a scalar or an array with loop-invariant subscript, the reduction is of type scalar, else it is an array reduction. One or several reduction statements may appear in a loop. The algorithm is shown in Figure 4.

5.3 Induction Variable Substitution

Figure 5 shows the induction variable recognition and substitution algorithm being developed in Cetus. The algorithm is applied to each loop nest, where it traverses three types of statements: (Ind) induction statements of the form `iv = iv + exp`, where `exp` is either loop-invariant or another induction variable, (Use) statements using the induction variable `iv`, and (Loop) inner loops using `iv` or containing induction statements for `iv`. The key equation in replacing induction variables in loop L is the assignment $val = initialval + inc_into_loop[L] + inc_after[stmt]$ of subroutine *Replace*; *initialval* is the value of the induction variable before loop L, $inc_into_loop[L]$ is the increment to the induction variable from before L to the beginning of the current iteration, and $inc_after[stmt]$ is the increment from the beginning of the iteration to statement *stmt*. All these values can be symbolic expressions; they are determined by subroutine $FindIncrements$.

The recognition of induction statements is similar to the recognition of reduction statements in Section 5.2. The algorithms handle generalized induction variables, where the increment *exp* can either be a loop-invariant expression or a linear expression that depends on another induction variable (with no cyclic relationships being allowed). For the latter case (i.e., *coupled induction variables*), the algorithm is successively applied to all induction variables, beginning with the one with no such dependences.

6 Application Passes

6.1 OpenMP-to-GPU: Automatic translation and compiler-driven optimizations

Hardware accelerators, such as GPUs, have emerged as powerful parallel platforms for high-performance computing. While a GPU provides an inexpensive, highly parallel system to application developers, its programming complexity poses a significant challenge for developers. Even though the CUDA (Compute Unified Device Architecture) programming model [18], recently introduced by NVIDIA, offers a more user-friendly programming model for General-Purpose computation on GPUs (GPGPU), programming GPGPUs is still complex and error-prone. In this project, we have developed an automatic OpenMP to CUDA GPU translator and optimization techniques. OpenMP is a well-established programming model for shared memory parallel computers. Due to the similarity between the OpenMP and CUDA programming models, we were able to convert the OpenMP parallelism, especially the loop-level parallelism, into the forms that best express parallelism in the CUDA programming model. However, there are architectural differences between traditional shared-memory machines, served by OpenMP, and stream architectures adopted by most GPGPUs, which may lead to the differences in optimization strategies. To address these issues, we have implemented several transformation techniques to optimize the performance of translated CUDA programs. Preliminary results in Figure 6 show that simple

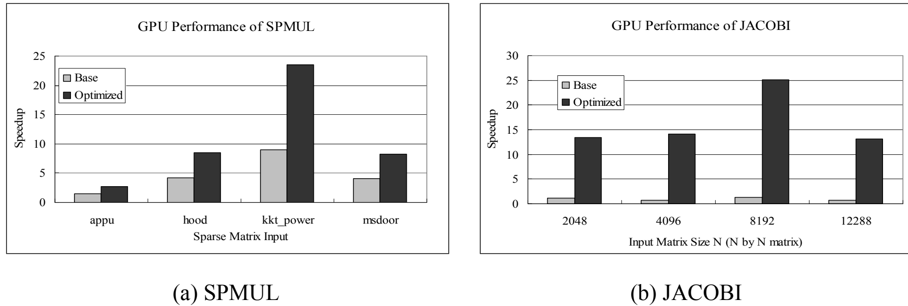
```
procedure SubstituteInductionVariable(iv,L,LIVE)
  input : iv, // induction variable to be substituted
          L // loop to be processed
          LIVE // set of variables that are live-out of L
  side effect : iv is substituted in IR
  inc = FindIncrement(L_0) // L_0 is the outermost loop of the nest
  Replace(L_0, iv)
  if (iv ∈ LIVE) InsertStatement("iv = inc") // at the end of L
end procedure

procedure FindIncrement(L)
  // Find the increments incurred by iv from the beginning of L:
  // - inc_after[s] is the increment from beginning of loop body to statement s
  // - inc_into_loop[L] is the increment from beginning of L to beginning of
  //   j^th iteration (j counts L's iterations from 1 to ub, in steps of 1)
  // - the subroutine returns the total increment added by L
  inc = 0
  foreach statement stmt in L of type Ind, Loop
    if stmt has type Loop
      inc += FindIncrement(stmt)
    else // statement has the form iv = iv + exp
      inc += exp
    inc_after[stmt] = inc
  inc_into_loop[L] = ∑_1^{j-1} inc // inc may depend on j
  return ∑_1^{ub} inc
end procedure

procedure Replace(L, initialval)
  // Substitute v with the closed-form expression
  val = initialval + inc_into_loop[L]
  foreach statement stmt in L of type Ind, Loop, Use
    if stmt has type Loop
      Replace(stmt, val)
    if stmt has type Loop, Ind
      val = initialval + inc_into_loop[L] + inc_after[stmt]
    if stmt has type Use
      Substitute(stmt, iv, val) // replace occurrences of iv in stmt with val
end procedure
```

Fig. 5. Induction Variable Substitution Algorithm. The algorithm handles generalized induction variables, as per [17]. The operators + and $\sum$ perform symbolic expression summations.

compiler transformation techniques, such as *loop interchange* and *loop coalescing*, can boost the performance of translated GPU programs. In case of *SPMUL* kernel translation (Figure 6 (a)), the base translation without any optimizations (*Base*) gives reasonable speedups, even though *SPMUL* is an irregular applica-



(a) SPMUL (b) JACOBI

Fig. 6. Performance comparison between base translated version (*Base*) and optimized version (*Optimized*). Speedups are over serial on CPU. The results show that even simple compiler translation techniques can boost the performance of translated GPU programs.

tion. However, after applying several optimizations, such as *local caching* optimizations exploiting registers or shared memory for frequently-accessed global data and *loop coalescing* transformation, the performance dramatically increases (*Optimized*). In the *JACOBI* case, naive translation (*Base*) degrades performance severely. However, simple *loop interchange* transformation can cure this problem (*Optimized*). More information about the translator and transformation techniques is shown in [19].

6.2 ATune: Compiler-Driven Adaptive Execution

The goal of this project is to develop compilers and runtime systems for dynamically adaptive applications. In previous work, we have created a system for dynamic adaptation of computational applications by tuning compiler options [20]. The current study investigates new methods to enable dynamic, adaptive optimization and tuning in diverse architectures, with emphasis on distributed irregular applications. For irregular applications, such as sparse matrix-vector (SpMV) multiplication kernels, static performance optimizations are difficult because memory access patterns may be known only at runtime. On distributed parallel applications, load balancing and communication cost reduction are two key issues. To address these issues, we have implemented a compiler-driven adaptive load-mapping and communication-algorithm-selection system. Our tuning system targets MPI-based distributed irregular applications, where the Cetus compiler inserts various algorithmic alternatives for given MPI collective communication calls, and generates necessary codes for adaptive iteration-to-process mapping. Actual tuning is conducted at runtime with the help of a Cetus-generated tuning driver.

Figure 7 shows the performance improvements of our adaptive runtime tuning system. We applied our techniques to distributed SpMV multiplication kernels (*SPMUL*). Experiments on 26 real sparse matrices in the UF Sparse Matrix

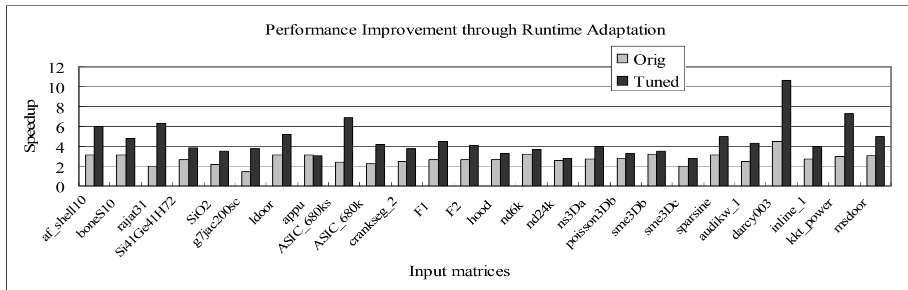


Fig. 7. Performance Improvement Through Adaptive Runtime Tuning. The bars show the speedups of the base parallel version (*Orig*) and adaptively tuned version (*Tuned*) on 16 nodes.

Collection [21] show that our adaptive tuning system (*Tuned*) reduces execution times up to 66.7% (33.3% on average) on 16 nodes. More detailed information can be found in [22].

7 Conclusions

Cetus has grown from a simple, student-designed source-to-source translator into a robust system that is now supported by the National Science Foundation as a community infrastructure. We have presented the current status in terms of the existing internal organization, analysis and transformation passes, and several applications. Cetus is ready for use by the user community. Via the Compunity Portal [7], we are able to respond to user requests and incorporate community-developed modules. Through these mechanisms, we expect Cetus to become a research infrastructure that is widely applicable to source-level optimizations and transformations for both multicore and large-scale parallel programs.

References

1. W. Blume, R. Doallo, R. Eigenmann, J. Grout, J. Hoeflinger, T. Lawrence, J. Lee, D. Padua, Y. Paek, B. Pottenger, L. Rauchwerger, and P. Tu, “Parallel programming with Polaris,” *IEEE Computer*, vol. 29, no. 12, pp. 78–82, Dec. 1996.
2. Seung-Jai Min, Seon Wook Kim, Michael Voss, Sang-Ik Lee, and Rudolf Eigenmann, “Portable compilers for openmp,” in *OpenMP Shared-Memory Parallel Programming*, Springer Verlag, Heidelberg, Germany, July 2001, Lecture Notes in Computer Science #2104, pp. 11–19.
3. Long Fei and Samuel P. Midkiff, “Artemis: practical runtime monitoring of applications for execution anomalies,” in *PLDI ’06: Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*, New York, NY, USA, 2006, pp. 84–95, ACM.
4. Ayon Basumallik and Rudolf Eigenmann, “Optimizing irregular shared-memory applications for distributed-memory systems,” in *PPoPP ’06: Proceedings of the*

eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming, New York, NY, USA, 2006, pp. 119–128, ACM.
5. Woongki Baek, Chi Cao Minh, Martin Trautmann, Christos Kozyrakis, and Kunle Olukotun, "The opentm transactional application programming interface," in *PACT '07: Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques*, Washington, DC, USA, 2007, pp. 376–387, IEEE Computer Society.
6. R. Asenjo, R. Castillo, F. Corbera, A. Navarro, A. Tineo, and E.L. Zapata, "Parallelizing irregular c codes assisted by interprocedural shape analysis," in *22nd IEEE International Parallel and Distributed Processing Symposium (IPDPS'08)*, 2008.
7. "Cetus: A Source-to-Source Compiler Infrastructure for C Programs [online]. available: http://cetus.ecn.purdue.edu," .
8. Troy A. Johnson, Sang-Ik Lee, Long Fei, Ayon Basumallik, Gautam Upadhyaya, Rudolf Eigenmann, and Samuel P. Midkiff, "Experiences in using Cetus for source-to-source transformations," in *Proc. of the Workshop on Languages and Compilers for Parallel Computing (LCPC'04).* Sept. 2004, pp. 1–14, Springer Verlag, Lecture Notes in Computer Science.
9. Randy Allen and Ken Kennedy, *Optimizing compilers for modern architectures*, Morgan Kaufman Publishers, San Francisco, CA, 2002.
10. Michael Wolfe, *Optimizing Supercompilers for Supercomputers*, MIT Press, Cambridge, MA, 1989.
11. William Pugh, "The Omega test: a fast and practical integer programming algorithm for dependence analysis," in *Proceeding Supercomputing '91.* 1991, pp. 4–13, IEEE Comput. Soc. Press.
12. William Blume and Rudolf Eigenmann, "The Range Test: A Dependence Test for Symbolic, Non-linear Expressions," *Proceedings of Supercomputing '94, Washington D.C.*, pp. 528–537, Nov. 1994.
13. William Blume and Rudolf Eigenmann, "Demand-Driven, Symbolic Range Propagation," *Lecture Notes in Computer Science, 1033: Languages and Compilers for Parallel Computing*, pp. 141–160, 1996.
14. Patrick Cousot and Radhia Cousot, "Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints," in *POPL '77: Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, New York, NY, USA, 1977, pp. 238–252, ACM.
15. Rudolf Eigenmann, Jay Hoeflinger, and David Padua, "On the Automatic Parallelization of the Perfect Benchmarks," *IEEE Trans. Parallel Distributed Syst.*, vol. 9, no. 1, pp. 5–23, Jan. 1998.
16. Peng Tu and David Padua, "Automatic Array Privatization," in *Proc. Sixth Workshop on Languages and Compilers for Parallel Computing, Portland, OR. Lecture Notes in Computer Science.*, Utpal Banerjee, David Gelernter, Alex Nicolau, and David Padua, Eds., August 12-14, 1993, vol. 768, pp. 500–521.
17. Bill Pottenger and Rudolf Eigenmann, "Idiom Recognition in the Polaris Parallelizing Compiler," *Proceedings of the 9th ACM International Conference on Supercomputing*, 95.
18. "NVIDIA CUDA [online]. available: http://developer.nvidia.com/object/cuda.html," .
19. Seyong Lee, Seung-Jai Min, and Rudolf Eigenmann, "OpenMP to GPGPU: A compiler framework for automatic translation and optimization," in *Proc. of the ACM*

Symposium on Principles and Practice of Parallel Programming (PPOPP'09). Feb. 2009, ACM Press.

20. Zhelong Pan and Rudolf Eigenmann, "Fast, automatic, procedure-level performance tuning," in *Proc. of Parallel architectures and Compilation Techniques*, 2006, pp. 173–181.
21. T. Davis, "University of Florida Sparse Matrix Collection [online]. available: http://www.cise.ufl.edu/research/sparse/matrices/," .
22. Seyong Lee and Rudolf Eigenmann, "Adaptive runtime tuning of parallel sparse matrix-vector multiplication on distributed memory systems," in *ACM International Conference on Supercomputing (ICS08)*, June 2008, pp. 195–204.