Open64-based Regular Stencil Shape Recognition in HERCULES *

Christos Kartsaklis   Oscar Hernandez

Computer Science Research Group
Oak Ridge National Laboratory
{kartsaklisc, oscar}@ornl.gov

Abstract

This paper discusses our design and implementation of a recognizer of stencil-like accesses in the Open64 compiler. This is a reference implementation in a real production compiler and serves as an example for supporting stencils on other compilers. We have developed the recognizer as part of our greater HERCULES framework which extends the Open64 compiler with user-level formulation of analysis and transformations. Our user base, which comprises computational scientists utilize this capability for identifying kernels that may benefit from a nested domain decomposition parallelizing scheme.

Categories and Subject Descriptors D3.4 [*Processors*]: Compilers

General Terms Stencil Code, Compiler

Keywords Open64, HERCULES, Array Accesses

1. Introduction

Stencil codes (nearest-neighbor) are a form of computation that occurs frequently in scientific applications that include structured grids as well as implicit and explicit partial differential equation (PDE) solvers, and in domains ranging from thermo/fluid dynamics, to climate modeling and electro-magnetics among others. An iterative explicit methods comprises a computationally intensive kernel. At each discrete time step, each point in the grid is updated using the previous timestep's data that its neighbors (in the grid) hold; Figure 1 depicts such a kernel.

One key characteristic of most stencil computations is the overlap in input values used to update multiple neighboring points. Exploiting this with the proper memory hierarchy is crucial in achieving good performance. Stencil computations continue to generate interest in the multicore era [5, 9–12, 14]. Due to their importance, they are often supported by Domain Specific Languages [2, 13, 15]. There is also growing interest in recognizing stencil kernels at the compiler level. The work in [7] uses an AST-based hierarchical pattern matching to identify kernels from a library of stencil templates. PADS [3], which is an OpenUH-based ([8]; an Open64-based compiler), uses a similar technique to eventually apply optimization strategies to stencils during GPU targeting. The Open64-based approach in [4] uses a new representation for loop nests and assignments to detect a set of commonly recurring kernels, including some stencils.

```
DO I=1,NY
  DO J=1,NX
    UNEW(I,J)=0.25*(
      UOLD(I-1,J) + UOLD(I,J+1) +
      UOLD(I+1,J) + UOLD(I,J-1))-
      H*H*F(I,J)
```

Figure 1. 2D Jacobi Sweep

The HERCULES [6] framework extends the Open64 compiler with user-level formulation of analysis and transformations. It extends Open64 with a new set of directives that users apply against actual language sources (C/C++/Fortran) in order to generate patterns. These patterns are of both a syntactical and performance nature (e.g. a parallelizable for-loop nested in conditional code). At the same time, it allows users to write transformation scripts that may be applied on any of the sources that the pattern matches. The transformation scripts consist of existing Open64 transformations (e.g. permutation) as well as in-house ones. The framework makes available the HERCULES-empowered Open64-derived compilers via a wrapping scheme – we offer, for instance, `hcc` as the C compiler and which is a wrapper around the HERCULES-aware `be` (back-end). These compilers integrate the pattern matching and transformation with the rest of the compilation process and are interchangeable with the Open64 `open*` compilers.

The example in Figure 2 demonstrates how HERCULES operates and also highlights how the stencil shape detection, which this paper discusses, can be incorporated in a HERCULES pattern. This pattern will match loop nests where the innermost contains a 5-point 2D stencil-like access against an array and where the array is read-only. Searching for the pattern in the figure via HERCULES against an input program, will prompt HERCULES to analyze the candidate loop (id `S`) and array for resemblance of a stencil code. In this paper we look at the algorithm that HERCULES uses to assess if and what kind of a stencil a given set of memory accesses is suggestive of.

The rest of the paper is organized as follows. We begin by giving a background on a set of Open64 data structures, then we describe

* The submitted manuscript has been authored by a contractor of the U.S. Government under Contract No. DE-AC05-00OR22725. Accordingly, the U.S. Government retains a non-exclusive, royalty-free license to publish or reproduce the published form of this contribution, or allow others to do so, for U.S. Government purposes.

WOSC '13 Novemember 2013, Indianapolis, USA

```
DO T=1,N
  DO J=JL,JU
! $hercules statement bind S
! $hercules    +S:body(B)
! $hercules    +B:stencil(ARRAY,5,2,[J,I])
! $hercules    +ARRAY:ronly()
    DO I=IL,IU
!     ...
    END DO
  END DO
END DO
```

Figure 2. A HERCULES pattern utilizing the stencil property

```
DO I=1,N
  DO J=1,M
    A(2*I,1+I+3*J) = ...
```

Figure 3. A loop stack of depth 2

how our algorithm operates, and conclude with our findings and some remarks on future work.

2. Relative Open64 Structures

Open64 ([1]) is a production-grade open-source compiler, with C, C++ and Fortran frontends, rich optimizing mid-ends, and backends for a few architectures. The Open64 intermediate representation (IR) is called WHIRL; the abstract syntax tree is a tree of WHIRL nodes (`WN`). Subroutines have an entry-point WN, and are associated with a Program Unit (`PU`). Open64 is organized as a series of phases through which the IR passes and gets "lowered" as the optimizations intensify and and target architecture nears. Our work occurs at the Loop Nest Optimizer (LNO) phase – a phase where the compiler provides structures and analysis to facilitate loop transformations. We give an overview of the core structure we use.

Open64 stores a summary of array accesses into instances of the `ARRAY_ACCESS` class, which, in turn, provides a summary on a per-dimension basis under instances of the `ACCESS_VECTOR` class. The `ACCESS_*`-related API is located in file `be/lno/access_vector.h`. The access vector allows one to test very quickly if an index expression is an affine expression of the governing loop stack's index variables, to test if the expression is constant, etc.

Consider the loop nest in Figure 3. In the rest of this paper, we will be referring to similar simple loop nests by the index variables in a stack-oriented way; the nest in the figure, for instance, will be called an "`I-J` loop stack". The access against array `A` is indeed an affine expression of loop index variables `I` and `J`; the per-dimension accesses have the canonical form, which is shown in Figure 4.

The `ACCESS_ARRAY` corresponding to these accesses will contain two `ACCESS_VECTOR` instances, accessible via `ACCESS_ARRAY ::Dim(UINT16)`, with minor dimensions appearing first; in our example, this would be the vector corresponding to `1+I+3*J`. Within each vector, the factors of the affine expression are encoded in `ACCESS_VECTOR:: _lcoeff` (the list of linear coefficients) in terms of the coefficient of each loop index variable. Open64 maintains a loop stack (the `DOLOOP_STACK`), with loop indices increasing as we descent the stack; in our example, the `I`-th loop is the first loop while the `J`-th loop is the second. As such, the first vector would contain coefficients $\{1, 3\}$, which would then be followed by vector with coefficients $\{2, 0\}$. The constant `1` in the `1+I+3*J` expression is held in the `ACCESS_VECTOR::Const_Offset` field.

$$0 + 2 \cdot I + 0 \cdot J$$
$$1 + 1 \cdot I + 3 \cdot J$$

Figure 4. Array accesses in canonical form

For each innermost `DO_LOOP` that appears in a simple `DO_LOOP`-only nest and each assignment-like statement:

1. Group array accesses by array id; for each group:
   (a) Initialize the set of candidate stencils.
   (b) Obtain the corresponding `STENCIL_ACCESS_ARRAY` instances; skip group if not possible.
   (c) Shrink candidates set.
   (d) Produce the `STENCIL_SHAPE_SUMMARY` and normalize.
   (e) Iterate the candidates looking for an exact match.

Figure 5. The main stencil shape extracting algorithm.

Note, also, of the convention of setting coefficients to zero to indicate that the respective index variable is not used in the expression.

3. Algorithm and Implementation

A very high-level view of the algorithm is shown in Figure 5. The algorithm operates by selecting a set of array load `WN`s, examining distances among index expressions and their relevance to the loop stack, and summarizing the gathered information.

We will now give a more low-level description. Starting from a given `PU`, the algorithm begins by identifying simple loop nests of `DO LOOP`s only. For each such loop, we have the right hand side (RHS) tree of all of its assignment-like `WN`s (`STID`, `ISTORE`, etc.) extracted from the body of the innermost loop. For each such RHS `WN`, we collect the set of "promising" array loads nested in it, i.e. loads that pass some of the qualifying criteria. These criteria include loads whose relative `ACCESS_ARRAY` suggests that bounds are not messy[1] and the only participant symbols in the index expressions draw symbols from the loop stack's index variable space (affine expressions). This selection, which is based on the RHS of assignments, is a heuristic in that the set could have as well been assembled from accesses scattered over the body of the `DO LOOP` that is being processed as opposed to the RHS of individual statements. Nonetheless, the `WN`s are then grouped according to the array they access.

We use our own array identifier, called the Array Unique Identifier, which is backed by structure `ARRAYUID`, for the following reason. While the symbol table index (`ST_IDX`) can be used for identifying symbols that are of an array type, this mechanism does not work well with composite types that contain arrays. For instance, a structure that has an array member nested in it, while it may be that variables of this structure are identifiable by symbol table entry, the nested field is somewhat anonymous (no symbol entry obviously), yet retrievable via the field operations.

We have created a new representation, the Stencil Access Array (SAA), which is backed by structure `STENCIL_ACCESS_ARRAY` and which is a compact summarization of an `ACCESS_ARRAY` aimed at stencil-oriented reasoning. The representation is optimized towards accesses which are characterized by index expressions that

[1] See `Bound_Is_Too_Messy(..)`

are affine expressions of single, non-scaled, loop variables with information recorded on a per-dimension basis. Assuming, for instance an `I-J-K` loop stack, the access `A(J,I-1,K-2)` satisfies the criterion since the index expression of each individual dimension is an affine expression of a single loop index variable; however, the access `A(I+J,J,K)` and `A(J,J,K)` fail to meet the criterion. We may write the summary for `A(J,I-1,K-2)` as $\{(1,0),(0,-1),(2,-2)\}$, i.e. as a list of stack-depth and displacement pairs. Information is recorded in a "WYSIWYG" manner, i.e. the left-most array index expression is the first in the list. A `STENCIL_ACCESS_ARRAY` is constructed using an `ARRAY`-operator `WN` as input.

In step 1 of the algorithm, the WHIRL nodes have been grouped by array id. We then attempt to obtain the SAA for each individual node. If this is not possible (e.g. if too messy), then the array is skipped. The SAA instances are maintained in a list. The first entry in the list dictates the order that the loop stack's index variables should appear in the next SAA instances. To clarify, assume an `I-J-K` loop stack, and two accesses: `A(I,J-1,K-1)` and `A(J,I,K-1)`. The corresponding SAAs are: $\{(0,0),(1,-1),(2,-1)\}$ and $\{(1,0),(0,0),(2,-1)\}$. When these two arrays are contrasted to each other, we find that stack-depth mismatch occurs (pair $(0,0)$ in the first SAA is inconsistent with the first pair of the second SSA, $(1,0)$ – different stack depths), and as such, array `A` has to be skipped. An array may be accidentally discarded if more than one stencil patterns are present in the same set of `WN`s that are being examined; this is discussed in the future work section.

We proceed now with a summarization phase that help us optimize the classification process and also spot further irregularities; we call this the "symmetry test". We examine how the displacements vary on a per dimension basis. Generally speaking, the goal is to tell whether a set of (unique) integer displacements $D \equiv \{d_1,...,d_n | d_i \in \mathbb{Z}\})$ is of the form $\{-s_1,...,-s_k,q_*,t_k,..,t_1\}$, such that for every $i \in \{1..k\}$, (1) $s_{i-1} - s_i = 1$ (if $k \geq 2$) (2) the difference $t_i - s_i$ stays constant. These properties ensure that there is some contiguity in how grid points are accessed. Item q_* is optional; when present, it must be the midpoint of s_1 and t_1, otherwise $t_1 - s_1$ must be an odd number. This later remark means that accesses are centered around some point, albeit that point is not necessarily accessed. Value k captures what potentially is the order of the stencil. The following sets, for instance, satisfy these criteria: (1) $\{-1,0,4,8,9\}$ suggests, for instance, that we may be dealing with an order 2 stencil, with accesses beginning 4 units away from the central point, which is likely accessed too; while on the other hand, (2) $\{4,5,6,8,9,10\}$ suggests an order 3 stencil, a single unit distance from the center, which is probably not accessed.

The cost of the test is mainly due to the sort operation; however, since set sizes scale by the number of accesses examined and these are few (generally the average number of syntax-level array accesses found in a loop body), this is not a concern. For each dimensions, its information can be expressed compactly by the triplet $(min(d_i), k, t_1 - s_1)$ in a 32-bit word, since these amounts are rather small and can be packed in smaller bitfields. Thus, for a set of n-dimensional accesses we will obtain n different summaries. Different tests on the summaries can give us an idea of what might be going on and bitwise operations can have surprising results. Assume a 3D array. If the last two fields of the triplets are the same across all summaries – something that can be tested with bitwise operations – this could be a cubic or spherical shaped stencil, with accesses beginning at a fixed distance from the center of the structure. However, had the last field varied and this would have signified an ovoid shaped stencil.

Let us define the `STENCIL_SHAPE_SUMMARY` as an object that "knows", i.e. encodes, where the points of the stencil lie on the grid; in our implementation, we have allocated a number of such objects for the stencils of interest, e.g. 7-point 3D heat transfer, 25-point finite difference, etc. A point is currently encoded compactly in a 32-bit word, providing $\lfloor 32/(n+1) \rfloor$ bits for nD coordinates – the additional bit is for the sign since points lie on both directions of the axes. The previous phase of the algorithm (steps b and c in Figure 5) helps us discard objects that mismatch grossly (number of points, dimensions, etc.). Given, now, the set of SAAs that we gathered earlier, we would like to convert them to an `STENCIL_SHAPE_SUMMARY` and compare this instance against our remaining candidates set. The comparison occurs by comparing the instances' sorted lists of points, but we have to do this for every candidate until we find a match. This comprises the classification process. To do so we need to normalize the offsets of the SSAs. The normalization happens by subtracting $min(d_i) + k_i$ from the i-th component of each coordinate. This is because $(min(d_1) + k_1, ..., min(d_n) + k_n)$ must be the midpoint of all SAAs in an nD space.

```
DO K=1,N-2
   DO J=1,N-2
      X(J+1,K+1) =
         H*(C1*U(J-1,K+1)+C2*U(J,K+1)-C3*U(J+1,K+1)+
            C2*U(J+2,K+1)-C1*U(J+3,K+1)) +
         H*(C1*U(J+1,K-1)+C2*U(J+1,K)-C3*U(J+1,K+1)+
            C2*U(J+1,K+2)-C1*U(J+1,K+3))
      DIFF = ABS(B(I+1,J+1) - A(I+1,J+1)) + DIFF
   END DO
END DO
```

Figure 6. A 9-point, 2-D, stencil.

4. Example

Assume the 2D stencil in Figure 6. The bounds have been intentionally shifted by 1 unit on both dimensions to highlight properties of the algorithm – one would usually expect a `J=2,N-1` and `I=2,N-1` for `(1:N,1:N)` arrays. Such a fragment would normally be part of a larger program with additional statements surrounding and interleaving this loop nest. We use Open64's infrastructure to identify simple loop nests such as the one above. The algorithm then proceeds by identifying assignment-like statements – two in this case. For each statement, the algorithm processes the RHS expressions. We will look at the first statement (`X(J+1,K+1)`) only. There is only one array used here, `U`; the algorithms records the 10 accesses against it. `K-J` is our "loop stack". The algorithm then examines how the two induction variables `K` and `J` are used. Every access i is of the form (`J+`α_i`,K+`β_i) (α_i and β_i are constants); this is what the algorithm expects. Being of the form (`K+`α_i`,J+`β_i) would be indifferent.

There are 9 unique accesses (`U(J+1,K+1)` is accessed twice), which can be written in terms of displacements as: $(-1,1)$, $(0,1)$, ..., $(1,3)$ – $(1,3)$, for instance, corresponds to the last access, `C(J+1,K+3)`. The next step of the algorithm is to summarize the per-dimension displacement information. The displacements for the first dimension (all displacements from `J`) are placed into a list, i.e. $\{-1,0,1,2,3,1,1,1,1,1\}$. The list gets deduplicated and sorted to yield $\{-1,0,1,2,3\}$. The algorithm concludes that the symmetry test has been passed and summarizes information into $(-1,2,2)$; -1 is the minimum value, 2 is the order, and 2 is the $t_1 - s_1$ difference. For the second dimension, the list $\{1,1,1,1,1,-1,0,1,2,3\}$ similarly reduces to summary $(-1,2,2)$. These are identical due to the "cross-like" access pattern. At this point, the classification process has to only focus on 9-point stencils, 2nd order, inclusive of accesses to the center point.

The algorithm subtracts the minimums from the last summaries, i.e. $(-1,-1)$ from all displacements, leading to displacements list $(-1,1)$, $(0,1)$, ..., $(1,3)$ being converted to $(0,2),(1,2),(2,2),(3,2),(4,2),(2,0),(2,1),(2,3),(2,4)$. Each such pair, gets packed into a 32-bit word (e.g. $(4,2)$ becomes `(4<<32)|2`), and the list gets converted to a list of integers. This is how an instance of `STENCIL_SHAPE_SUMMARY` looks like. The classification concludes by iterating the classifiers, which are themselves instances of `STENCIL_SHAPE_SUMMARY` too, until an instance carrying an identical list has been identified.

5. Conclusions and Future Work

The purpose of this paper was to discuss our effort towards a compiler-based analysis for recognizing what stencil shape a set of memory accesses resemble. We have implemented the analysis and supporting structures under our Open64-backed HERCULES framework, which groups different tools for detecting patterns in programs. We presented a set of complementary Open64 data structures, namely the `STENCIL_ACCESS_ARRAY`, `ARRAYUID` and `STENCIL_SHAPE_SUMMARY`, which contribute towards a stencil shape identification framework.

We plan to add support for periodic accesses. Given, for instance, an array `A(1:N)`, we want to regard the accesses set `A(I)`, `A(I+N-2)` and `A(I+2)` as a *periodic* stencil shape of stride 2, i.e. identify the occurrence of a wraparound. We are also working to improve our `WN` selection heuristic which is currently limiting the approach in that it is only array access operators under the same expression tree that can only be considered as stencil shape defining. The algorithm also is quite strict in that the group of array accesses that are being examined must much the classifier exactly; if after the classification there are still a few accesses remaining unpaired, the classification is aborted.

Acknowledgments

This research used resources of the Oak Ridge Leadership Computing Facility at the Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725.

References

[1] Open64 Compiler. `http://www.open64.net/`, 2013.

[2] M. Christen, O. Schenk, and H. Burkhart. PATUS: A Code Generation and Autotuning Framework for Parallel Iterative Stencil Computations on Modern Microarchitectures. In *Parallel Distributed Processing Symposium (IPDPS), 2011 IEEE International*, pages 676 –687, may 2011. doi: 10.1109/IPDPS.2011.70.

[3] D. Han, S. Xu, L. Chen, and L. Huang. PADS: A Pattern-Driven Stencil Compiler-Based Tool for Reuse of Optimizations on GPGPUs. In *Parallel and Distributed Systems (ICPADS), 2011 IEEE 17th International Conference on*, pages 308 –315, dec. 2011. doi: 10.1109/ICPADS.2011.94.

[4] J. He, A. Snavely, R. Van der Wijngaart, and M. Frumkin. Automatic Recognition of Performance Idioms in Scientific Applications. In *Parallel Distributed Processing Symposium (IPDPS), 2011 IEEE International*, pages 118 –127, may 2011. doi: 10.1109/IPDPS.2011.21.

[5] S. Kamil, C. Chan, L. Oliker, J. Shalf, and S. Williams. An Auto-Tuning Framework for Parallel Multicore Stencil Computations. In *Parallel Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pages 1 –12, april 2010. doi: 10.1109/IPDPS.2010.5470421.

[6] C. Kartsaklis, O. Hernandez, C.-H. Hsu, I. Ilsche, J. Wayne, and R. L. Graham. HERCULES: A pattern driven code transformation system. In *17th International Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS) held in conjunction with the 26th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, Shanghai, China, May 2012.

[7] C. W. Keßler. Pattern-driven automatic parallelization. *SCIENTIFIC PROGRAMMING*, 5:251–274, 1996.

[8] C. Liao, O. Hernandez, B. Chapman, W. Chen, and W. Zheng. OpenUH: an Optimizing, Portable OpenMP Compiler. *Concurrency and Computation: Practice and Experience*, 19(18):2317–2332, 2007. ISSN 1532-0634. doi: 10.1002/cpe.1174. URL `http://dx.doi.org/10.1002/cpe.1174`.

[9] T. Lutz, C. Fensch, and M. Cole. Partans: An autotuning framework for stencil computation on multi-gpu systems. *ACM Trans. Archit. Code Optim.*, 9(4):59:1–59:24, Jan. 2013. ISSN 1544-3566. doi: 10.1145/2400682.2400718. URL `http://doi.acm.org/10.1145/2400682.2400718`.

[10] J. Meng and K. Skadron. Performance Modeling and Automatic Ghost Zone Optimization for Iterative Stencil Loops on GPUs. In *Proceedings of the 23rd international conference on Supercomputing*, ICS '09, pages 256–265, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-498-0. doi: http://doi.acm.org/10.1145/1542275.1542313. URL `http://doi.acm.org/10.1145/1542275.1542313`.

[11] D. Orozco, E. Garcia, and G. Gao. Locality Optimization of Stencil Applications Using Data Dependency Graphs. In *Proceedings of the 23rd international conference on Languages and compilers for parallel computing*, LCPC'10, pages 77–91, Berlin, Heidelberg, 2011. Springer-Verlag. ISBN 978-3-642-19594-5. URL `http://dl.acm.org/citation.cfm?id=1964536.1964542`.

[12] A. Schafer and D. Fey. High Performance Stencil Code Algorithms for GPGPUs. *Procedia Computer Science*, 4(0):2027 – 2036, 2011. ISSN 1877-0509. doi: 10.1016/j.procs.2011.04.221. Proceedings of the International Conference on Computational Science, ICCS 2011.

[13] A. Solar-Lezama, G. Arnold, L. Tancau, R. Bodik, V. Saraswat, and S. Seshia. Sketching stencils. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, pages 167–178, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-633-2. doi: 10.1145/1250734.1250754. URL `http://doi.acm.org/10.1145/1250734.1250754`.

[14] M. Wittmann, G. Hager, and G. Wellein. Multicore-Aware Parallel Temporal Blocking of Stencil Codes for Shared and Distributed Memory. In *Parallel Distributed Processing, Workshops and Phd Forum (IPDPSW), 2010 IEEE International Symposium on*, pages 1 –7, april 2010. doi: 10.1109/IPDPSW.2010.5470813.

[15] C.-K. L. Yuan Tang, Rezaul Alam Chowdhury and C. E. Leiserson. Coding Stencil Computation using the Pochoir Stencil-Specification Language. In *HotPar '11: 3rd USENIX Workshop on Hot Topics in Parallelism*, Berkeley, California, May 2011.