A Stencil Compiler for the Connection Machine Model CM-5

Ralph G. Brickner, Kathy Holian
Balaji Thiagarajan, S. Lennart Johnsson

CRPC-TR94457
June, 1994

Center for Research on Parallel Computation
Rice University
P.O. Box 1892
Houston, TX 77251-1892

A Stencil Compiler for the Connection Machine Model CM–5

Ralph G. Brickner [1], Kathy Holian, Balaji Thiagarajan[2]
Los Alamos National Laboratory
S. Lennart Johnsson[3]
Thinking Machines Corporation

Abstract

In this paper we present the design of a stencil compiler for the Connection Machine system CM–5. The stencil compiler will optimize the data motion between processing nodes, minimize the data motion within a node, and minimize the data motion between registers and local memory in a node. The compiler will natively support two–dimensional stencils, but stencils in three dimensions will be automatically decomposed. Lower dimensional stencils are treated as degenerate stencils. The compiler will be integrated as part of the CM Fortran programming system. Much of the compiler code will be adapted from the CM–2/200 stencil compiler, which is part of CMSSL (the Connection Machine Scientific Software Library) Release 3.1 for the CM–2/200, and the compiler will be available as part of the Connection Machine Scientific Software Library (CMSSL) for the CM–5.

In this paper, we report on the implementation status of the stencil compiler. In particular, we discuss optimization strategies and status of code conversion from CM–2/200 to CM–5 architecture, and report on the measured performance of prototype target code which the compiler will generate.

1 Introduction

This paper discusses design goals, performance of prototype code, and implementation status of a "stencil compiler" for the Connection Machine system CM–5. A "stencil" for the purpose of this paper is a weighted sum of circularly shifted arrays. A specific example expressed in the CM–Fortran language (CMF) [6] is

```
REAL, DIMENSION( NX, NY ) :: DST, SRC, C1, C2, C3, C4,  C5

DST = C1 * CSHIFT( SRC, DIM=1, SHIFT=-1)  +
```

[1]Presenting Author. Address: C-3, MS-B265, LANL, Los Alamos, NM 87545. E-Mail: rgb@lanl.gov. Phone: 505-667-8385 (voice), 505-665-5220 (FAX).
[2]Also with the Department of Computer Science, Syracuse University
[3]Also affiliated with the Division of Applied Sciences, Harvard University

```
&      C2 * CSHIFT( SRC, DIM=1, SHIFT=+1)  +
&      C3 *         SRC                    +
&      C4 * CSHIFT( SRC, DIM=2, SHIFT=-1)  +
&      C5 * CSHIFT( SRC, DIM=2, SHIFT=+1)
```

which represents a second order accurate discretization of the Laplacian operator on a regular grid in two dimensions. In the example, "DST" is the *destination* array, "SRC" the *source* array, and "C1" through "C5" are the *coefficient* arrays; they are in this example arrays, but they may be scalar variables also. Because there are shifts in two dimensions of the source, this is a *rank 2* stencil. In general, our stencil compiler can have multiple sources arrays, and even multiple destination arrays. The number of coefficient arrays are determined by the size of the stencil, i.e., the number of grid points that are convolved. In practice, the stencil compiler in its first incarnation will only handle a number of coefficients compatible with the number of registers in a vector unit. If there are more coefficients than what can be handled in the register file, then the compiler reverts to executing the CMF stencil subroutine compiled with the standard CMF compiler.

Operations like the one above occur with great frequency in scientific computing. Any simulation which solves partial differential equations by finite difference methods is likely to incorporate stencil operations at an "inner loop" level. Image filtering and compression may consist largely of stencils, and other examples abound.

It is therefore desirable to be able to execute stencil operations efficiently. Thinking Machines Corporation developed an experimental "convolution compiler" for the CM–2 that won the Gordon Bell Award in 1990 [2]. Expanding upon that work, a CM–2/200 stencil production quality stencil compiler for the Connection Machine Scientific Software Library was developed in collaboration between Thinking Machines Corp. and Los Alamos Laboratory [1].

The next section discusses the principles of stencil optimization, suggesting areas for improved performance compared to the standard CMF compiler. The following sections discuss the implementation of those optimizations in prototype codes for the CM–5, and the delivered performance. The codes have been developed for automatic generation by a compiler, envisioned as a pre–processor for the CM–5 CMF compiler. The architecture of this proposed compiler is discussed, along with a comparison with the existing stencil compiler for the CM–2/200 in the Connection Machine Scientific Software Library (CMSSL). Finally, we discuss the implementation status of the CM–5 stencil compiler, and draw some conclusions.

2 Principles of Stencil Optimization

In CM Fortran, multiple array elements are assigned to each physical processor. Currently, the data distributions are always block distributions. The *subgrid* assigned to each processor consists of consecutive array elements in each dimension. The proper

focus of our work is thus the calculation of an entire subgrid of stencil results, not a single stencil element. There are three primary areas in which our compiler may gain in performance over the CM Fortran compiler:

1. Full utilization of the interconnection network in doing multidimensional and bidirectional communications required by most stencils
2. Elimination of memory–to–memory moves occurring as part of the `CSHIFT` communications intrinsics used to implement stencils in CM Fortran
3. Optimal Floating Point Unit (FPU) register use in the computational part of the stencil calculation.

The first point we leave up to the authors of the CM Fortran runtime system [8], except for elimination of overhead due to multiple communications calls at the CM Fortran level. The `CSHIFT` intrinsics in the CM Fortran expression for a stencil are atomic processes, in that a call to `CSHIFT` requires that all data motion be finished before the call returns, and the potential for optimization across multiple calls is thus lost. Our solution to this problem in the stencil compiler is to call a single routine which accomplishes the *minimum* required data motion between processors, by calling a runtime communications function as discussed in a later section. This procedure provides a high percentage of the potential communications bandwidth possible in stencil communications.

The second area for performance improvement consists of eliminating the memory–to–memory move component of the `CSHIFT` intrinsic function. `CSHIFT`, as currently implemented, "knows" nothing about the use of the result array it produces. Therefore, it is constrained to follow the definition of its functionality in the language specification, which states that *all* the elements of the source array are shifted in the index space. However, if the only use of the `CSHIFT` result is in the calculation of a stencil on an FPU's subgrid, we can more cleverly implement this shift functionality by leaving in place all the array elements which would have moved *only* within the FPU's memory, and adjusting our memory address calculations for these elements accordingly. Because stencils normally involve a significant number of `CSHIFT` calls, and because the memory–to–memory move cost increases with the subgrid size, this can result in significant savings in execution time for large subgrids and large stencils.

The third and final optimization we discuss in this section consists of reusing the stencil source array elements once they are loaded into the FPU registers. For concreteness, we consider the standard five–point stencil in two dimensions, as commonly encountered in discretizing the Laplace operator in two dimensions, and written out in the introduction. As seen in Figure 1, we can combine multiple instances of the stencil along one axis to obtain a *multistencil*. This axis will be referred to as the *pipeline* axis, which we take for the sake of discussion to be axis 1. The number of stencils combined along the pipeline axis is the *multiwidth*. The benefit of a multistencil is that, for example, the rightmost point of the leftmost stencil requires loading exactly

the same source array element as the center point of the next stencil to the right. For the multistencil of multiwidth eight, we need to load 26 source array elements, compared to the naive count of 40 obtained from loading all the source array elements for each stencil calculation. The greater the width of the stencil along the pipeline axis, the greater is the savings in memory loads for a given multiwidth. Stencils may also be combined along the other stencil axis of a five–point stencil to obtain further gains in register reuse. We refer to this axis as the *sweep* axis. Register reuse through combining stencils along the sweep axis is obtained by loading only the leading edge of the multistencil as it sweeps through the source array along the sweep axis. For this to work, we must store in the FPU registers as many rows of source data (along the pipeline axis) as the stencil is wide along the sweep axis.

So far, we have discussed only two–dimensional stencils, implicitly embedded in two–dimensional data arrays. In this case, the two axes over which we re–use data exhaust the possibilities. However, for higher rank stencils, there is the possibility of optimizing register usage over other dimensions. For example, in the three–dimensional analog of the five–point stencil discussed above, we could load in multiple layers of data in the third axis, just as we build a multistencil along the pipeline axis. In practice, this approach is limited in usefulness by the number of FPU registers available for storing the source data. Even if there are enough FPU registers to fit the extended stencil, it is likely the vector length would be relatively short. This would restrict the efficiency of the stencil calculation, since a vector length of eight is required for peak efficiency. Because of these considerations, we choose to optimize our stencil calculations over only two axes. We then loop the two–dimensional calculations through the remaining axis of the three–dimensional data, and so on for higher rank data.

One significant complication in the implementation of the multistencil scheme is the presence of *wings* as shown in Figure 1. The wing elements are, in general, not part of the main source array subgrid, but instead may come from edge arrays obtained from neighboring processors. Because they potentially belong to a different array than the central part of the multistencil, they require their own address and stride registers. Since in many processor architectures, the size of the register set is fairly limited, the number of wings in a multistencil is one severely limiting factor on the size of the stencils the compiler can accommodate. The solution to this problem in our proposed design for the CM–5 stencil compiler is to allocate a "ghost source" array with the same machine geometry as the user's source array, but with "ghost cells" around the boundary which are to be filled with data from physically neighboring processors. This requires one extra memory copy (from the original source to the ghost source), but greatly simplifies the code required to perform the arithmetic, and also eliminates extra address and stride registers required for the wings, when the wings are in separate edge arrays.

The optimization principles described above for the design of the CM–5 stencil compiler are the same as for the CM–2/200 stencil compiler [1]. Therefore, much of the stencil compiler algorithmic development port from the CM–2/200 to the CM–5.

3 Stencil Compiler Implementation Strategy

In this section, we describe the strategy for implementation of the stencil optimization principles described in the preceding section.

An early yet crucial decision for the CM–5 stencil compiler was to provide a separate source array, one for each of the input source arrays, with "ghost cells" on the boundary. These extra ghost cells are to be filled in by data from off–processor grid positions. (This type of array is known as an "overlap array" in the compiler community). The ghost cells allow greatly simplified code generation, and reduce the number of registers required for addresses and strides of the edge data.

Given the general approach of using ghost arrays for the stencil source data, the following major tasks must be accomplished in order to calculate a stencil:

1. Allocate ghost array memory.
2. Fill the ghost arrays from the appropriate source arrays (on–processor data motion).
3. Retrieve the edge data into the ghost cells on the boundary of the ghost arrays.
4. Loop over subgrid axes.
5. Calculate the arithmetic operations involved in the stencil.
6. Store the result.
7. Cleanup.

Each ghost array must have the same distributed (physical) geometry as the corresponding source arrays and destination arrays passed into the stencil subroutine. Each ghost array must also have the same subgrid shape, but with extra ghost cells around the edges, of width equal to the stencil width in that direction. To simplify all the following operations, it is also desirable to convert the two–dimensional (for example) geometry of the input and output arrays into a four–dimensional geometry, such that the first two axes are purely on–processor (subgrids), and the second two are purely physical (off–processor). This is accomplished by means of the CMF alias package, which provides a comprehensive set of routines to recast a given array geometry into another array geometry, without moving any of the array data. A side effect of the construction of these aliases is that we obtain all the information we need to construct the ghost array. Because the physical part of the geometry requires one element per physical processor, the optimized stencil code we are developing requires compilation with the CMF "–nopad" switch [7]. All the array aliasing is done in CMF code, in the subroutine which the user calls. The ghost array memory is allocated with CMF code, called from this same top–level stencil routine. This setup code is called only once for a given set of actual input arguments, to reduce overhead.

Once we have set up and allocated ghost arrays, we need to fill their interior with the data from the source arrays. Because of the ghost cells along the boundary, the interior data for a ghost are not contiguous (although they are contiguous in the source array). Therefore, a simple block copy of the entire array cannot be done. We have explored three methods for carrying out this copy of data: CMF code, custom CDPEAC code [5], and calls to the runtime system routine "CMCOM_array_section_transfer" [8]. Based upon timing studies, we have chosen the CDPEAC for this phase of the target code. We anticipate a small library of pre–written routines which can be called for various stencil/array geometries, each of which contains a small amount of interface code, and encapsulates a call to a CDPEAC kernel. The compiler will then generate a call to the appropriate interface routine, with the proper parameters for the user arrays as determined at runtime. The CDPEAC routine does all looping over the subgrid axes.

The next step is to perform the physical communications required to move data between physical processors. "Corners", when required, are moved with the "column" edges after the "row" edges have been filled in. We have explored two methods for carrying out this move of data: CMF code, and calls to the run-time system routine "CMCOM_physical_rotate" [8]. Based upon timing studies and ease of implementation, we have chosen the CMCOM routine for this phase of the target code. As for the interior fill phase, we anticipate a small library of pre–written routines which can be called for various stencil/array geometries, each of which contains a small amount of interface code, and encapsulates a call to "CMCOM_physical_rotate". The compiler will then generate a call to the appropriate interface routine, with the proper parameters for the user arrays as determined at run–time. The CMCOM routine does all looping over the subgrid axes.

Looping over the two array axes selected for register reuse optimization is done by the arithmetic portion of the stencil code, as described below. However, we must also loop over the other axes of the user arrays, in the case that those arrays are of higher rank than two. This looping structure is to be done by "C" code generated at (stencil) compile time, since we know at that time the ranks of the stencil and the user arrays.

The arithmetic calculations in our prototype code are carried out in CDPEAC [5], a C–like programming language which allows both C constructs for control and scalar data, and direct specification of vector instructions for the FPUs. The CDPEAC code does all looping over the pipeline and sweep axes; it also chooses the maximum multistencil (vector length) possible for the remaining extent along the pipeline axis, for every swath along the sweep axis through the user arrays. For each swath, a number of "groups" of instructions are required, corresponding to the width of the stencil along the sweep axis. These groups are required in order to cycle the loaded source data through the registers, reusing the data in each of several steps along the sweep axis. For our canonical five–point example, there are three groups, with a given vector of data participating first as the top row (vector) of points, then the middle row, then the bottom row. The following step along the sweep axis frees up those registers, and the new leading edge of data are loaded into them. There is initialization code before each swath along the sweep axis, to load in the initial rows of source data. And there is

additional code at the termination of the sweep axis loop to account for array extents along that axis which are not a multiple of the number of groups. When the sweep–axis loop is executing, the code performs the following steps: 1) load the leading edge of source data into the current FPU registers corresponding to the "top" row of the multistencil; 2) load a vector of coefficients for the first term in the stencil; chain these data into the initial product; 3) load a vector of coefficients for the second term of the stencil; chain these data into the second product, add the result of the previous product (a single vector "mult–add" instruction); 4) continue until all terms are calculated; 5) store the result.

The store of results is straightforward for double precision results. However, for single precision results, there is a performance penalty on the CM–5 architecture. This can sometimes be alleviated if the results to be stored correspond to single precision words which are contiguous in memory. Because a stencil may span any three dimensions of any allowable CMF array, there is, in general, no guarantee that the optimization strategies described above will result in the calculation of contiguous single precision words. Therefore, we intend to investigate alternative vectorization strategies which would allow us to calculate contiguous single precision results, and store them as a single vector.

Because the stencil compiler generates code to allocate CM memory for the ghost arrays, we anticipate providing a mechanism to deallocate that memory when it is no longer needed. Alternatively, if future timing studies indicate the performance penalty is acceptable, we might deallocate the ghost array memory upon exit of the stencil code.

4 Measured Performance for Stencil Components and Prototypes

The specific implementation strategies discussed above were the result of testing various approaches to the different components of the prototype stencil code. In the following sections, we present some of the results of those tests for components of five– and nine–point stencils in two–dimensions (the nine–point stencil is the relative of the five–point stencil displayed in the introduction, but with four "corners" added). All tests were run on CM–5 systems at Thinking Machines Corporation during March–April of 1994, with the floor CM Fortran compiler (CMF 1.2.1-2), the floor runtime system (Version 8.0), and the "–O" optimization flag for the CM Fortran compiler. C code was compiled with "gcc", using the "–O4" optimization flag; CDPEAC code was compiled with the "–O2" optimization flag. The timings were done on systems running CMOST Version 7.3. In the following, we present only the *elapsed* time, not the CM Busy time, because the elapsed time is the actual time a user needs to wait until the problem completes. All times are for a timing loop of 1000 iterations. In general, we plot execution time as a function of subgrid length, for a square subgrid. That is, for a subgrid length of 4, we are timing performance on 1000 iterations of a process, on a 4×4 subgrid. Finally,

because all timings were done on busy time-sharing systems, we took the *minimum* elapsed time for a number of runs.

We tested three approaches to filling the interior of the ghost array: Straight CM Fortran code, custom CDPEAC code, and calls to CMCOM_array_section_transfer. The CM Fortran code was:

```
      real*8, array( 0:NSX+1,0:NSY+1,1:NPX,1:NPY ) :: ghost_src
      real*8, array( 1:NSX,  1:NSY,  1:NPX,1:NPY ) :: src

      ghost_src( 1:NSX, 1:NSY, :, : ) = src( 1:NSX, 1:NSY, :, : )
```

Here, as with the following, the original two–dimensional arrays have been aliased to four–dimensional arrays by means of the CM Fortran ALIAS package, so that, e.g., NSX is the number of elements in the subgrid along the x–direction. The third and fourth axes are purely physical, i.e. non–local.

Figure 2 presents the results of comparing the CMCOM_array_section_transfer call with the CDPEAC call for a variety of *square* subgrids. We do not show the CMF timings, since they are so large that plotting all three methods on a single graph makes the two non–CMF methods appear identical in performance. Typically, the CMF time is an order of magnitude (or more) longer for a given subgrid length. As is evident, the best result is obtained from the custom CDPEAC code, which is the method we have chosen for the stencil compiler.

We tested two approaches to filling the edges of the ghost array: Straight CM Fortran code, and calls to CMCOM_physical_rotate. The CM Fortran code evolved through a number of versions; we present the final version here. Both CM Fortran and CMCOM versions depend upon the ghost array already being filled with interior data. This avoids "cross–geometry" moves, which can have a significant performance penalty. The CM Fortran code was (with the appropriate declarations):

```
      real*8, array( 0:NSX+1,0:NSY+1,1:NPX,1:NPY ) :: ghost_src

CC    Recall e.g., the array section ghost_src(0,1:NSY,:,:) is *3D*
CC                                                 ^    ^ ^
CC                                                 1    2 3

      ghost_src(    0,1:NSY,:,:) =
     &  CSHIFT( ghost_src(NSX,1:NSY,:,:),    DIM=2, SHIFT=-1 )

      ghost_src(NSX+1,1:NSY,:,:) =
     &  CSHIFT( ghost_src(  1,1:NSY,:,:),    DIM=2, SHIFT=+1 )
```

```
 ghost_src(0:NSX+1,    0,:,:) =
&   CSHIFT( ghost_src(0:NSX+1,NSY,:,:), DIM=3, SHIFT=-1 )

 ghost_src(0:NSX+1,NSY+1,:,:) =
&   CSHIFT( ghost_src(0:NSX+1,  1,:,:), DIM=3, SHIFT=+1 )
```

The comments indicate why, when we actually want to communicate along the third dimensions of the original array in the first and second statement, the CSHIFT argument is for dimension "2". Figure 3 presents the results of comparing this method with the other two methods for a variety of *square* subgrids. As is evident, the best result is obtained from the CMCOM_physical_rotate routine, which is the method we have chosen for the stencil compiler.

Because the CM Fortran compiler does not reuse registers in the manner described in our discussion of optimizations, it is difficult to compare between CM Fortran arithmetic code and our code. However, our timing studies have indicated our CDPEAC code is competitive with CM Fortran code for a similar operation – that is, for every grid point, load of a single source value, followed by a sum of products of that value with nine different coefficients, i.e.

```
 dst = c1*src + c2*src + c3*src + c4*src + c5*src +
&      c6*src + c7*src + c8*src + c9*src
```

The results are shown in Figure 4, where we see our CDPEAC code is indeed competitive with the CMF code for the above operations on small subgrids. However, for large subgrids, the CDPEAC does loose somewhat; we conjecture this is becasue the maximum vector length is not 16, but 14 for our calculations, due to the existence of the stencil "wings". Once, again, we stress that these are *not* the same calculations, but are presented to give some idea of the efficiency of our CDPEAC code.

Figures 5 and 6 present the comparison between CMF code for the nine–point and five–point stencils, and the stencil compiler prototype code. As is evident, the prototype code provides significantly better performance for large subgrids. The five–point CMF code is that displayed in the introduction. The "naive" nine–point CMF code is similar, except it has four additional terms, each a doubly–nested `CSHIFT` operation. The "reduced nine–point" results [4] refer to a method of calculating the nine-point stencil which involves only four `CSHIFT` calls instead of the naive twelve `CSHIFT` calls, but with the cost of adding four temporary arrays. This method also requires pre-shifting the coefficient arrays, which is acceptable when the stencil is employed in an inner loop (typically an iterative solver), and the coefficients are calculated outside the loop. As is evident, the stencil compiler prototype code provides significant performance improvement for large subgrids.

5 Implementation Status

The anticipated structure of the CM–5 stencil compiler will comprise 1) a separate compiler step, callable from CM Fortran, and consisting of separate parser and code generator passes; and 2) a set of library routines for performing array–filling and communications functions, as well as some auxiliary routines to assist in executing the required looping structures. In terms of use, the user will need to break out the desired stencil into a separate subroutine, complete with declarations (but not requiring shape or size information). This subroutine must be placed in a file with a ".stencil" extension, and passed to CM Fortran with the usual "-c" flag for compilation to a ".o" file. The result, if everything works out, is an apparently normal ".o" file which the user then links with the other files in the application. This general scheme is the same as the CM–2 stencil compiler present in CMSSL Release 3.1 for the CM–2 [9].

Implementation of the CM–5 stencil compiler will involve reuse of a great deal of code from the CM–2 stencil compiler. The parser is unchanged, expect for additional refinement to enhance reliability. All the analysis code in the compiler (for example, to determine the number of register groups, register assignments, etc) will be directly usable also. The major modification to the compiler code will be in the code generation phase. The CM–2 stencil compiler generated calls to custom microcode [1]; the CM–5 compiler will generate calls to the above–described CDPEAC and CMCOM routines and also generate portions of the CDPEAC code and all the appropriate wrappers. Nevertheless, we anticipate much of the structure surviving. Currently, we have begun building the initial parts of the code generation phase as we have done our timing studies to determine our optimization strategies. This initial code-generation phase creates the highest–level CM Fortran code, callable from the user code.

6 Testing and Verification of the Stencil Compiler

The potential "space" of possible stencils, even considering only rank–three and lower stencils, is enormous. The goal of the stencil compiler is to support asymmetric patterns, number of points limited only by register availability, single and double precision data, scalar or array coefficients, and multiple source and destination arrays. Not only is the stencil itself greatly variable, but it may be embedded in arrays of any rank allowed by CM Fortran. Furthermore, given a stencil embedded in arrays of a given rank, the axes of the arrays may be distributed or local, and the specific shape of the distributed and local parts is not known until runtime, and varies from machine size to machine size.

Because the space of possible stencils is so large, we are from the outset designing sophisticated, automated testing procedures to ensure we explore large areas of the possible space of stencil parameters. We have taken the automated testing procedures of David Kramer as the starting point for this work [3]. Furthermore, we intend to implement regression testing during the development process to help ensure quality

control.

The automated testing procedure is a systematic approach developed to test and validate the functionality of the stencil compiler. We have used a semi-exhaustive technique to generate array geometries to test the compiler. The motivation for using this technique is as follows

1. A large subset of possible geometries can be tested
2. Performance results on "odd" geometries can provide useful feedback for improving performance of the compiler.

In outline, the automated testing routine functions as follows. We refer the reader to the CM–5 CM Fortran Performance Guide [10] for a discussion of arrays geometries.

1. Define the inputs provided by the user that tester will use to generate test cases
2. Define the public interface to CMF routines for checking correctness
3. Define the public interface to the stencil compiler–generated routines
4. For each input line
   (a) Check the validity of inputs based on certain preset rules
   (b) Generate processor masks, hence forth called "pmasks". The pmasks are generated using the highest axis varying fastest ordering (HVFO). We have also used the lowest axis varying fastest ordering(LVFO) which will generate a different set of pmasks. The test set will be made more comprehensive in future by using more innovative ordering techniques.
   (c) Generate a set of permutation for the pmask. Each permutation is called an array geometry. For each array geometry:
      - Allocate arrays for the cmf interface and stencil interface
      - Call the cmf interface that produces the cmf_result
      - Call the stencil interface that produces the stencil_result
      - Compare stencil_result and cmf_result. If the stencil compiler routine works correctly these two results should be the same.

During the course of the stencil compiler development, we intend to extend and refine these techniques further.

7 Conclusions

The stencil compiler prototypes we have presented here illustrate methods which can provide significant performance gains over ordinary CM Fortran code for a wide variety of stencils. The techniques for gaining this performance are neither simple nor

straightforward, from user perspective, without a great deal of aid from software tools. The compiler we are presently developing provides those tools, and as part of CMSSL, can significantly enhance the productivity of CM–5 users.

8 Acknowledgements

This work was performed in part under funding through DOE TTI CRADA LA93C10070, and the DOE HPCCI Program. CM–5 access was provided by Thinking Machines Corporation and the Advanced Computing Laboratory of Los Alamos National Laboratory. We would like to thank the following people for helpful conversations and assistance: Palle Pederson, Kapil Mathur, David Kramer, Steve Heller, Bob Lordi, Mark Bromley (TMC), and Rick Smith (LANL).

References

[1] Ralph G. Brickner, William George, S. Lennart Johnsson, and Alan Ruttenberg. A stencil compiler for the Connection Machine Models CM–2/200. In *Proceedings of The Fourth International Workshop on Compilers for Parallel Computers*, pages 68–78. Delft University of Technology, December 1993.

[2] Mark Bromley, Steven Heller, Tim McNerney, and Guy L. Steele. Fortran at ten Gigaflops: The Connection Machine convolution compiler. In *Proceedings of ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*, pages 145–156. ACM Press, June 1991.

[3] Thinking Machines Corporation David Kramer. Automated testing routines for cmssl, 1994. Private Communication.

[4] Los Alamos National Laboratory Richard Smith. Stencil subroutines for the lanl climate model codes, 1994. Private Communication.

[5] Thinking Machines Corp. *CM–5 VU Programmer's Handbook, CMOST Version 7.2*, 1993.

[6] Thinking Machines Corp. *CM Fortran Reference Manual, Version 2.1*, 1993.

[7] Thinking Machines Corp. *CM Fortran Release Notes, Version 2.1*, 1993.

[8] Thinking Machines Corp. *CM Run–Time System Architectural Specification, Version 7.2*, 1993.

[9] Thinking Machines Corp. *CMSSL for CM Fortran: CM–2/200 Edition, Version 3.1*, 1993.

[10] Thinking Machines Corp. *CM–5 CM Fortran Performance Guide, Version 2.1*, 1994.

Figure 1: Multistencils

x = a point for which stencil is calculated (destination).
One source array element is required for each grey block in the *multistencil*.
One coefficient is required for each grey block in each *stencil*.

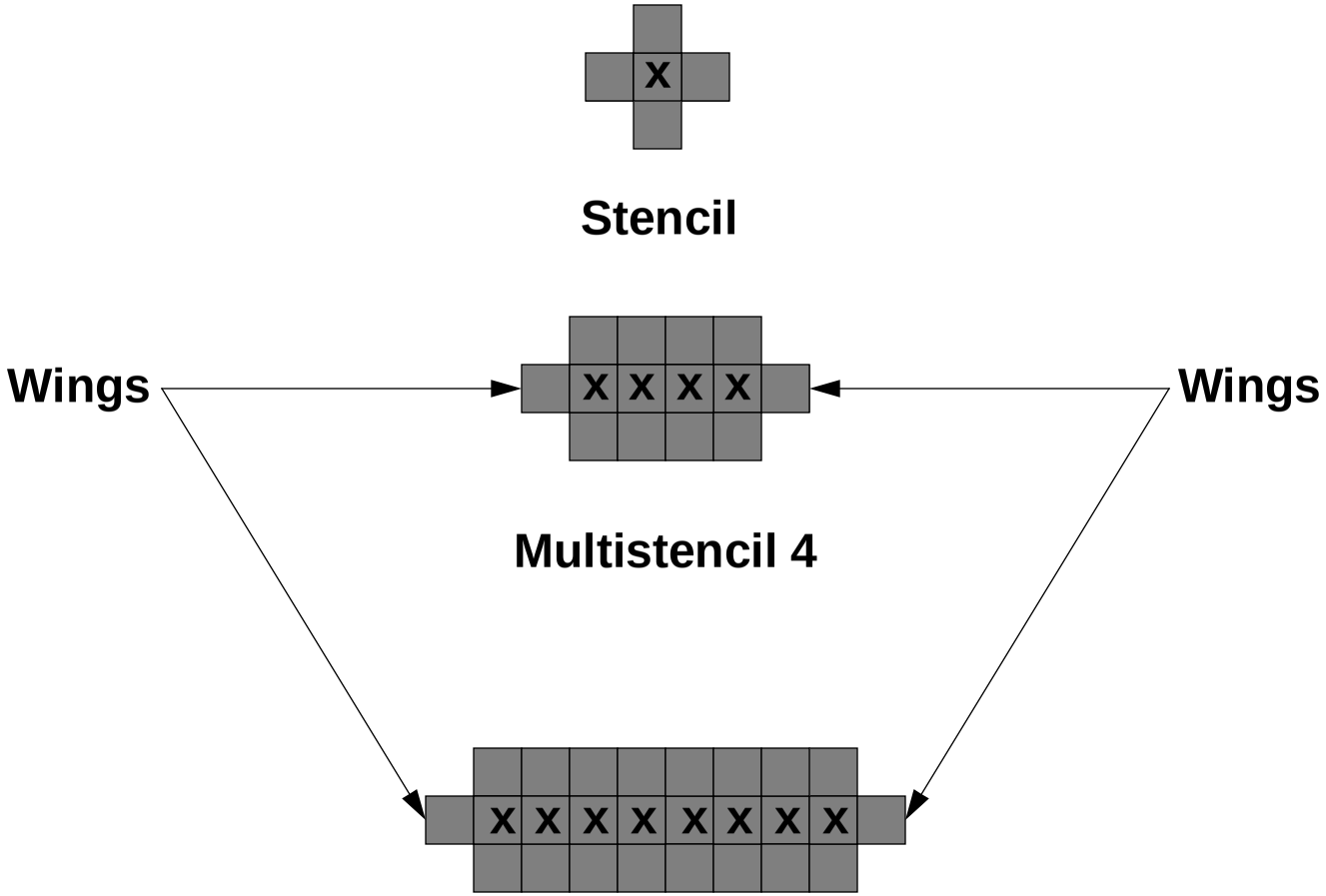

Figure 2. CMCOM/CDPEAC Ghost Interior Fill

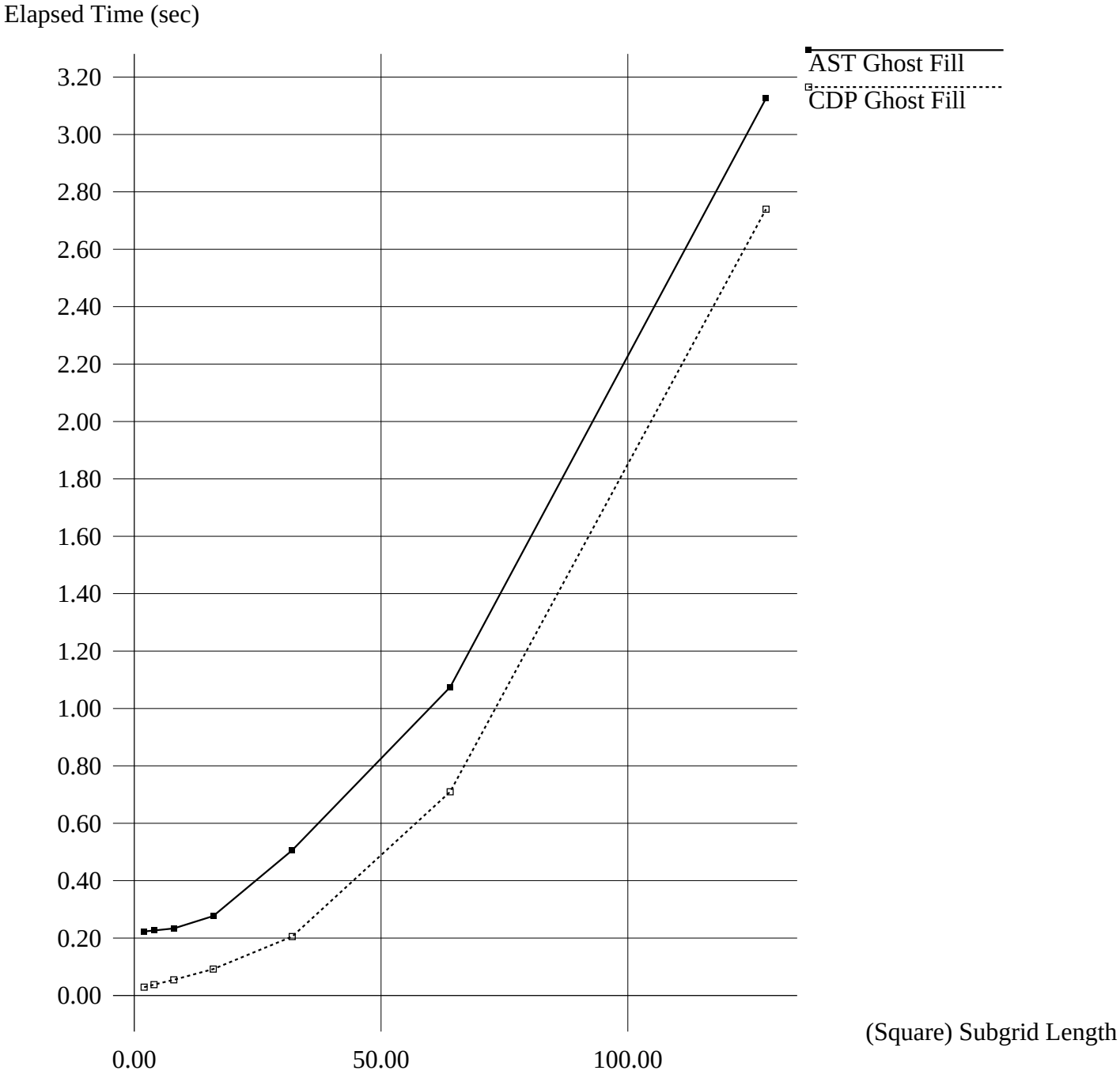

Figure 3. CMF/CMCOM Physical Rotate

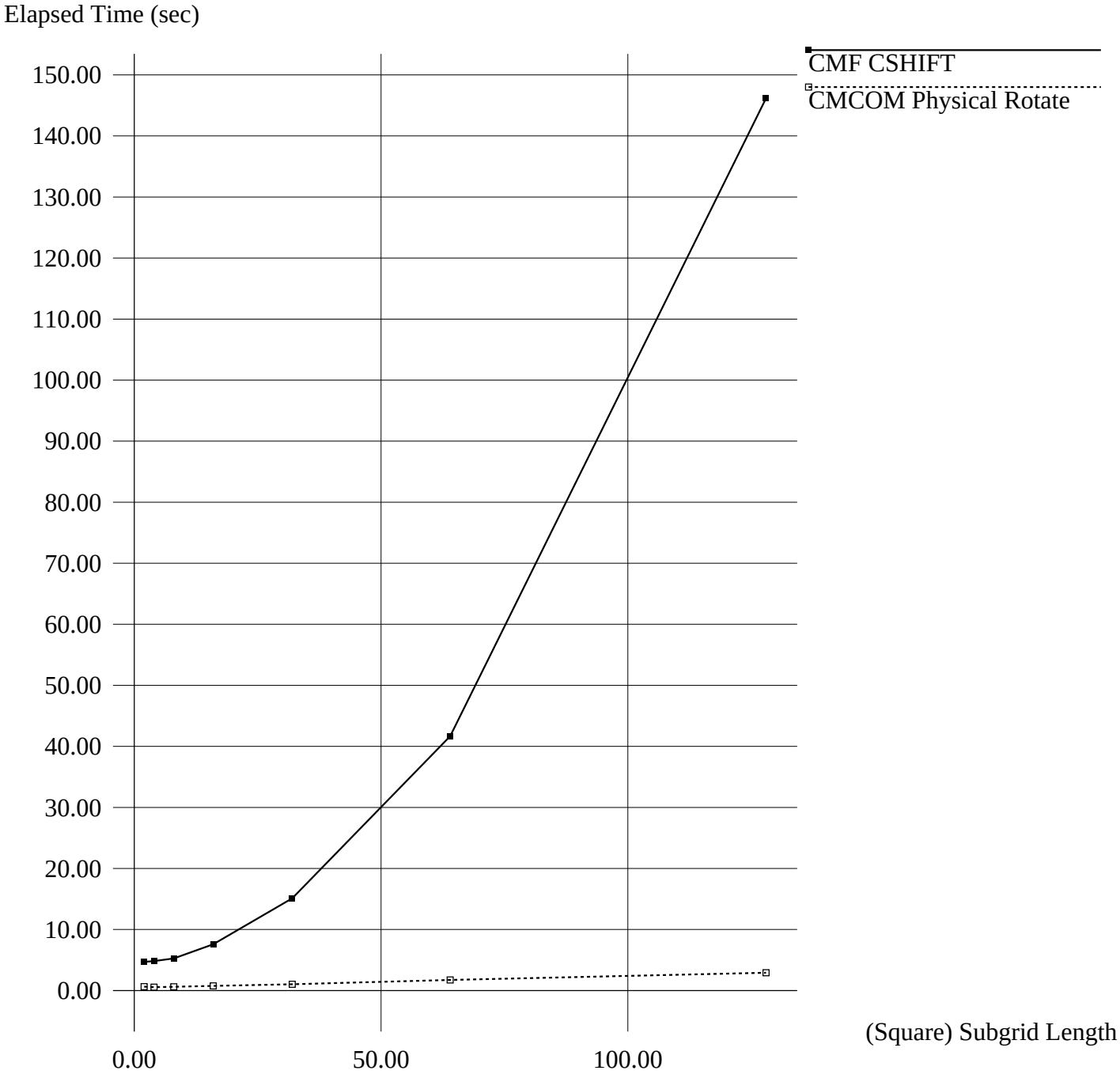

Figure 4. CMF/CDPEAC Arithmetic

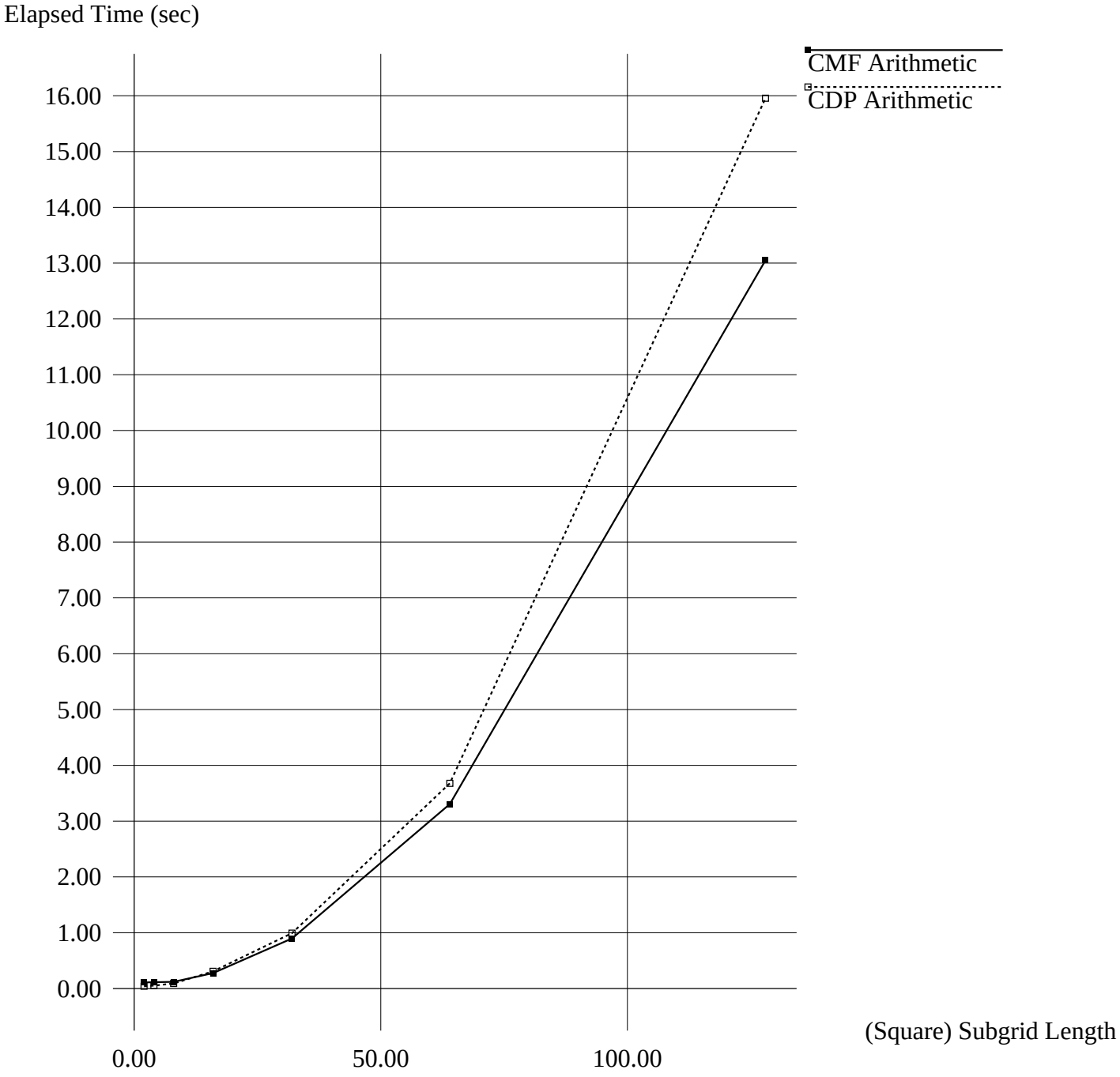

Figure 5. Two Dimensional, 5pt Stencils

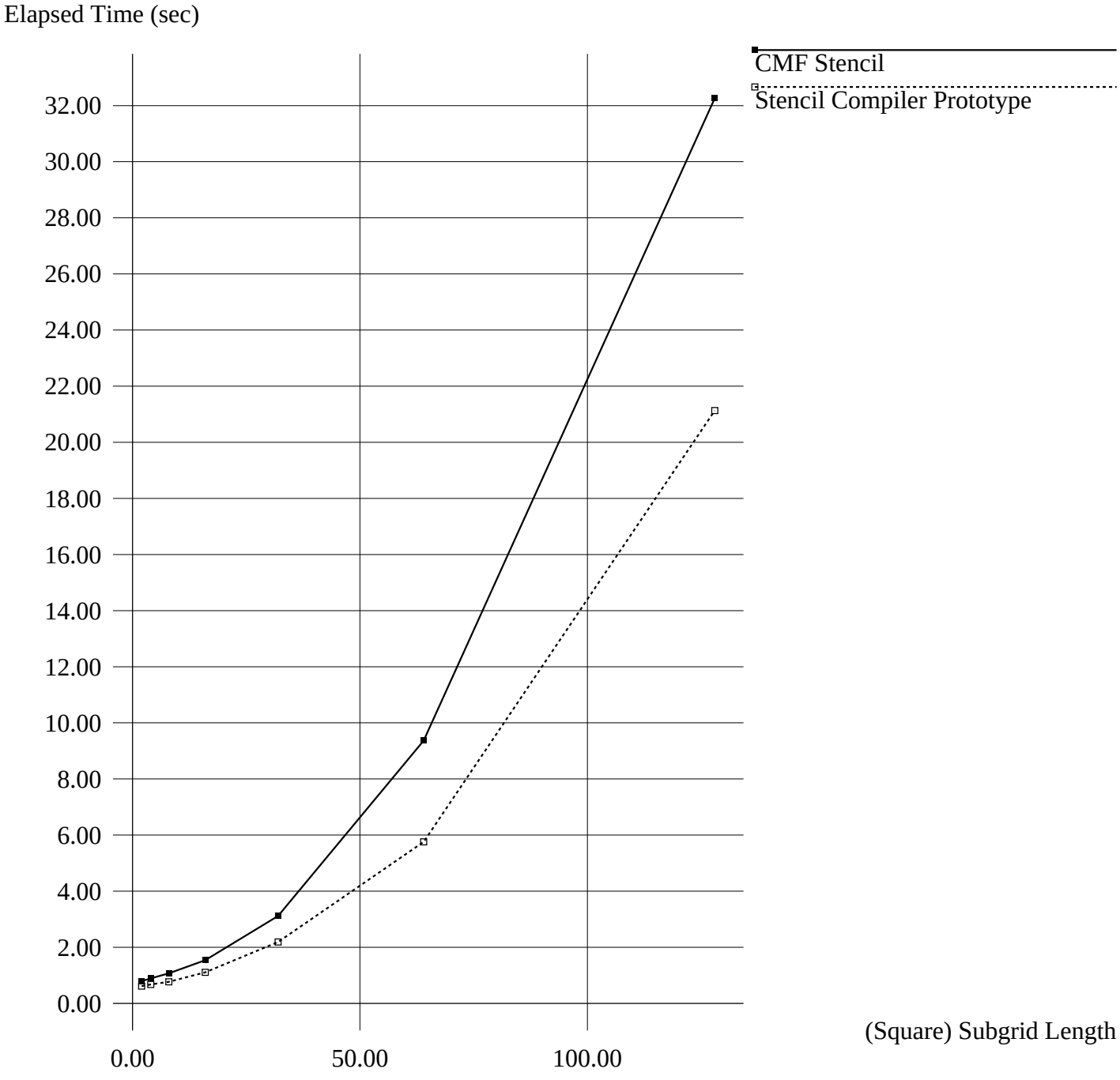

Figure 6. Two Dimensional, 9pt Stencils