### Citation

### Citable link
[http://nrs.harvard.edu/urn-3:HUL.InstRepos:27031377](http://nrs.harvard.edu/urn-3:HUL.InstRepos:27031377)

### Terms of Use
This article was downloaded from Harvard University’s DASH repository, and is made available under the terms and conditions applicable to Other Posted Material, as set forth at [http://nrs.harvard.edu/urn-3:HUL.InstRepos:dash.current.terms-of-use#LAA](http://nrs.harvard.edu/urn-3:HUL.InstRepos:dash.current.terms-of-use#LAA)
A Stencil Compiler for the Connection Machine Models CM-2/200

Ralph G. Brickner
William George
S. Lennart Johnsson
Alan Ruttenberg

TR-22-93
November 1993

Parallel Computing Research Group
Center for Research in Computing Technology
Harvard University
Cambridge, Massachusetts

A Stencil Compiler for the
Connection Machine Models CM–2/200

Ralph G. Brickner
Los Alamos National Laboratory

William George
Los Alamos National Laboratory

S. Lennart Johnsson
Thinking Machines Corporation

Alan Ruttenberg
Thinking Machines Corporation

Abstract

In this paper we present a Stencil Compiler for the Connection Machine Models CM–2 and CM–200. A stencil is a weighted sum of circularly-shifted CM Fortran arrays. The stencil compiler optimizes the data motion between processing nodes, minimizes the data motion within a node, and minimizes the data motion between registers and local memory in a node. The compiler makes novel use of the communication system and has highly optimized register use. The compiler natively supports two-dimensional stencils, but stencils in three or four dimensions are automatically decomposed. Portions of the system are integrated as part of the CM Fortran programming system, and also as part of the system microcode. The compiler is available as part of the Connection Machine Scientific Software Library (CMSSL) Release 3.1.

1 Principles of Stencil Optimization

The computational environment of the Connection Machine includes support for the virtual machine model to provide for efficient utilization of the processor Floating Point Unit (FPU). In this model, multiple virtual processors are assigned to each physical processor, or, in CM Fortran (CMF) [5] terminology, multiple Fortran array elements are assigned to each physical processor. For the current CMF compilers, data distributions are always block distributions, so that the subgrid assigned to each processor consists of consecutive array elements in each dimension.

1 Currently with the Department of Computer Science, Clemson University

2 Also affiliated with the Division of Applied Sciences, Harvard University
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 implementation of a stencil calculation:

1. Full utilization of the binary cube interconnection of the processors 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 CMF

3. Optimal FPU register use in the computational part of the stencil calculation.

A side benefit of the last point is more efficient pipeline constructions than those used by CMF. These points for performance enhancement were first discussed in [1] in the context of a research stencil compiler. The rest of this section discusses these points in more detail.

The CM−2/200 architecture includes a binary cube interconnection topology among the FPU’s. The CMF compiler (more precisely its geometry manager, part of the compiler runtime system) can exploit this topology to ensure that neighboring array elements are assigned to physically neighboring FPUs. The CMF LAYOUT directive can be used to specify :NEWS distribution, which results in the Gray−coding [3, 4] of the CMF array onto the binary cube topology. The CMF intrinsic CSHIFT results in communications between nearest−neighbor FPUs for arrays layed out with the :NEWS directive.

For production−sized machines, and stencils of limited rank (1−4), enough physical binary cube channels exist to allow communications along different array axes for the stencil to be assigned to distinct communication links. Details of simultaneous communications in multiple dimensions is discussed in [2], which describes the CMSSL [6] PSHIFT functionality. The CSHIFT intrinsics are atomic processes, in that a call to CSHIFT requires that all data motion be finished before the call returns, and the potential for simultaneous communications along multiple dimensions is thus lost. Our solution to this problem in the stencil compiler is to call a set of PSHIFT−like communications primitives to accomplish data motion between neighboring FPUs concurrently. These primitives provide 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 FPUs subgrid, we can more cleverly implement this shift functionality by leaving in place all the array elements which would have moved only within the FPUs 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 depends upon the subgrid size, this can result in significant savings in execution time for large subgrids and large stencils.

The 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. 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. 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 require 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. Furthermore, 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.

In our implementation, the pipeline axis of the multistencil behaves much like a vector length, thus the name pipeline axis. Moreover, much of the stencil compiler algorithmic development will transfer to other machines with vector units, such as the CM–5. 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.

## 2 Stencil Compiler System Architecture

The CM–200 Stencil Compiler comprises three major software components: customized microcode, a compiler to generate calls to the microcode and supply the precise register and memory access patterns required, and a run–time system which, 1) resolves some data distribution issues unknown at compile time, and 2) applies the microcode to a set of user arrays at run–time time. We discuss these components in more detail below.
2.1 The Stencil Microcode

The stencil microcode resides in the CM–200 microcode store on the sequencer, which is the component of the machine that receives high-level commands from the front-end via the “FIFO” (First-In, First-Out channel) and translates them into micro-sequenced instruction streams to be executed on the CM–200 processing units. The microcode to support stencil calculations was developed under the assumption that all communications operations necessary to calculate the stencil for an entire subgrid have been done prior to calling the microcode. (We discuss this process below in the Run–Time System subsection.) The basic microcode entity which performs the stencil calculations applies a two-dimensional swath through the user’s CMF arrays. One axis of this swath corresponds to the pipeline axis. Within the multiwidth, register sharing is fully exploited. The other axis of the swath corresponds to the sweep axis. Along this axis, the swath extends to the mid-point of the user array. Register sharing along the sweep axis is also fully exploited. The swath is illustrated in Figure 2. By sweeping only half of the source/destination subgrid at a time, it suffices to take boundary conditions into account only at the beginning of a sweep. This simplifies code generation. However, it also means that the Stencil Compiler is limited to stencils whose height in the sweep axis is at most one half the height of the source/destination subgrid along the sweep axis. Otherwise, stencil points near the midpoint of the subgrid would “run off” the subgrid on the other side.

The stencil microcode calculates an entire half-swath in one call. Thus, call overhead is minimized, and pipeline length is maximized. During the sweep of the half-swath, for every multiwidth-wide row of destination elements to be calculated, the center of the leading edge of the required source elements are first loaded into the FPU registers. Then, the wing elements of the leading edge are loaded, followed by loading the coefficients for each individual stencil as the multiply–accumulate instructions are issued. When an entire multistencil has been calculated, it is stored to the destination addresses. All address registers are adjusted to point to the next row of data, and the process is repeated until the end of the half-swath is reached.

The register allocation for the source and destination registers, and the association of these registers with the loading of coefficient elements in the proper order, is done by the code generator at compile time. The result is a sequence of dynamic instructions, which specify the exact FPU registers to be used for loads, stores, and multiply–accumulate operands. These dynamic instructions are loaded from front–end memory into the CM sequencer scratch memory (SRAM) at stencil computation initialization; the microcode reads the specific dynamic instructions from SRAM during execution in order to get the correct register usage pattern.

During the course of the sweeping process, the FPU registers assigned to a given row in the multistencil cycle through a pattern depending on the multistencil height along the sweep axis. This cycle is accommodated by generating a number of groups of dynamic instructions, with this number being determined by how many rows in the sweep axis are required for the same register access pattern to repeat. For the five–point stencil
illustrated in the Figures, the number of groups is three.

In addition to routines that evaluate half-swaths, there are supporting routines that, for example, load the FPU registers prior to beginning the sweep. The routines load either multistencil elements one at a time from the bottom edge or the source/destination subgrid, or individual non-leading wing elements.

All the microcode routines expect certain arguments to be passed in as part of the dynamic instruction sequence, and certain arguments to be pre-loaded into sequencer registers. Array addresses and strides, for example, are all expected to be pre-loaded into certain sequencer registers. Other degrees of freedom, for example the multiwidth, are handled by providing separate routines for the different values.

There are separate microcode routines to support single-precision coefficients and data, single-precision coefficients and double-precision data, and double-precision coefficients and data. The computational process described above is designed for optimization of stencils with array-valued coefficients. Scalar coefficients are handled by simply storing the scalar values to CM memory and setting the coefficient memory strides to zero.

Finally, the Stencil Compiler Run-Time System (SCRTS) is responsible for applying the half-swath operation to the entire subgrid, by looping over the pipeline axis, and repeating the process for both up- and down-sweep (both half-swaths along the sweep axis).

### 2.2 The Compiler

The compiler, which supports the above microcode, is a standalone front-end executable which is called from CMF. To use the Stencil Compiler, the programmer places a stencil expression into a separate file with a .stencil extension, and runs the CMF compiler as usual. CMF recognizes this special file-name extension and runs the Stencil Compiler on the file, producing an apparently normal .o file, which is linked as usual. In actuality, the object file produced contains memory representations of the dynamic instructions for the microcode, Internal Macro Procedure (IMP) wrappers, calls to the Stencil Compiler Run-Time System (SCRTS), and a version of the user’s original CMF stencil code. This code is called if the SCRTS detects a situation it cannot handle.

The compiler decodes the user’s input file using a parser derived from lex and yacc. The input file syntax is a small and specific subset of CMF, including subroutine statements, array and scalar declarations, compiler directives, comments, and the stencil expression itself, which must be a sum of terms, each of which is a coefficient multiplying a shift expression. A shift expression is either a CMF array, or a series of (possibly) nested CSHIFTs. If nested, no dimension/distance pairs may be repeated. Multiple source arrays are allowed within a stencil expression, multiple stencil assignments are allowed within a stencil subroutine, and multiple stencil subroutines are allowed within the stencil file.
The parser produces an intermediate data file which incorporates all relevant information extracted from the stencil file. The compiler reads this intermediate file and computes a number of stencil characteristics used in the code generation phase, such as the number of register groups and information requests to be passed to the SCRTS. The code generation phase attempts to generate code for multiwidth 8, 4, 2, and 1 multistencils, subject to instruction timing, and FPU and sequencer register constraints. Separate code is generated for upsweeps and downsweeps to fully accommodate non-symmetric stencils. A C-language wrapper is generated which contains the interface between the user’s CMF code and the SCRTS. Finally, a number of language processors are called to process the generated source files, including “cc”, “cmf”, “impass”, and “ld”. The result is a linkable object file, which needs to be linked with the SCRTS (included in the CMSSL library) to resolve external references.

2.3 Run–Time System

The SCRTS is responsible for a number of tasks postponed to run–time, as well as performing the necessary communications operations and sweeping the basic multistencil calculations through the FPU subgrid. Because the array geometry is not known until run–time, subgrid axes extents, strides, and addresses are not known to the compiler, and cannot be used to make optimization decisions. The current SCRTS does not make optimization decisions at run–time, except for choosing the maximum multiwidth for the current half–swath.

The compiler–generated bridge routine between the user’s CMF code and the SCRTS includes a large structure containing a great deal of information about the stencil geometry, as well as the expected geometries of the CMF arrays passed in as actual arguments to the stencil subroutine. One large part of this interface is a set of “FIFO push descriptors”, which instruct the SCRTS which values are to be pushed down the FIFO into sequencer registers after a microcode computational routine is begun. Some of these values are known at compile time (for example the number of “wings” in the stencil), but most are determined at runtime from the CMF array descriptors.

When the bridge routine calls the SCRTS, an extensive set of checks is made to ascertain whether the SCRTS is able to perform the requested calculation on the actual arguments. If not, an error code is returned to the bridge routine, which then calls the user’s original CMF subroutine. Reasons for an error return include actual arguments which do not match dummy arguments in the stencil subroutine, CMF arrays with “garbage masks”, machine configurations which do not support the required PSHIFT–like communications, and subgrids too small or of odd shape. The design goal of the Stencil Compiler is to accept all stencils as inputs and to always return correct answers even when in fact it is not used, but the users CMF code is invoked. Thus, by structuring a user code for the Stencil Compiler, enhanced performance is obtained whenever it can be used, while a correct result is always guaranteed.

The general strategy of the SCRTS, after checking its input, is to set up edge arrays
for data coming in from other FPU(s), perform the multidimensional communications
required by the stencil, and then strip–mine the stencil calculation through the subgrid.
While the stencil rank may be 0, 1, 2, 3, or 4, on any set of not necessarily contiguous
CMF array axes, the data and coefficient arrays to which the stencil is to be applied
may be any size, shape, and layout allowed by CMF, except that \texttt{SEND} ordered axes are
not allowed in any dimension spanned by the stencil shape itself. Specifically, \texttt{SERIAL}
axes are fully supported.

When the SCRTRTS has successfully completed the requested stencil calculation, it re-
turns to the bridge routine without an error code; the bridge routine immediately
routines to the user’s CMF routine.

\section{Performance}

Performance of the stencil compiler is highly dependent upon a number of factors,
including number of stencil points, stencil size and shape, data type, coefficient type
and rank, and the actual CMF arrays passed in as arguments. One crucial factor is
whether a given stencil will “fit” into the FPU and sequencer register set at all, and
if so, what the maximum multiwidth is. As a general consideration, the greater the
multiwidth, the better the performance. If even a multiwidth one multistencil cannot
be made to fit into the register set, the Stencil Compiler returns with an informative
message. The user is then free to attempt to decompose the stencil into smaller stencils
by hand. Future work will address automatic methods for such decomposition.

There are a number of other considerations which can be used as a general guideline.
Arrays with garbage masks will result in calling CMF to perform the calculation, so no
performance gain is to be expected in this case. When the stencil compiler performs
the calculation, we expect

\begin{itemize}
\item a gain over CMF for high–rank stencils, since the Stencil Compiler performs
\texttt{PSHIFT}–like communications on all dimensions of the stencil at once
\item a gain over CMF for large subgrids, since the Stencil Compiler eliminates the
on–processor component of \texttt{CSHIFT};
\item a gain over CMF for large subgrids, since it can then use large multiwidths, and
amortize pipeline overhead over more array elements
\item a gain over CMF for double–precision data, since the reduced number of source
loads is emphasized in this case;
\item a loss over CMF for scalar coefficients, since the Stencil Compiler strategy wastes
memory bandwidth in this case.
\end{itemize}

Figure 3 provides a graph of a performance comparison between CMF and the Stencil
Compiler on the five–point stencil in two dimensions. Runs were done on square
subgrids of length 2, 4, 8, 16, and 32. The stencil used array-valued coefficients, and double-precision coefficients and data. The codes were compiled with CM–2 CMF Slicewise 2.0 Beta 2 compiler, and the CMSSL 3.1 Stencil Compiler, SCRTS, and microcode. They were run on a CM–2 at Thinking Machines Corporation on a Sun 4 front-end under timesharing (with, however, no other users). Times are for 100 calls to the stencil calculation.

References


Figure 1: Multistencils

- One coefficient is required for each grey block in each stencil.
- One source array element is required for each grey block in the multistencil.
Figure 2: Sweep and Pipeline Axes

Sweep Axis

Pipeline Axis

Upward

Downward

Starting Position

$\frac{-3}{2}$
Figure 3: 5-Pt Double-Precision Stencil, Array Coefs

CM Time for 100 Iterations (sec)

Subgrid Length

CM Fortran
Stencil Compiler