Automatic Code Generation for DDM-VM[1] in GCC Using GRAPHITE: A Field Report

[1]Petros K. Panayi, [2]Zbigniew Chamski, [1]Samer Arandi, [1]George Michael, [1]Paraskevas Evripidou

[1]Department of Computer Science, University of Cyprus, Nicosia, Cyprus
{petrosp,samer, xeirwn,skevos}@cs.ucy.ac.cy
[2]Infrasoft IT Solutions, Płock, Poland
zbigniew.chamski@gmail.com

Abstract: The Data-Driven Multithreading (DDM) is a non-blocking multithreading execution model based on the data-flow execution paradigm. The DDM-VM is a virtual machine implementation of DDM targeting both homogeneous and heterogeneous multicore processors. In this paper, we report on our work in automatic DDM-VMs code generation using GCC, utilizing the GRAPHITE framework as the main dependence analysis and transformation engine, and demonstrating that a DDM thread-centric parallelization pass can be built incrementally upon the existing GCC infrastructure. We outline the steps for extending our implementation to achieve a better support of the data-flow execution model, and propose a series of extensions to the GRAPHITE framework.

1 Introduction

The shift to multicore processors has reignited the compiler research on automatic parallelization. The GNU Compiler Collection (GCC) community, following the need for implicit parallel code generation for multicores, developed GRAPHITE – a loop-centric program analysis and transformation framework based on the polyhedral program representation [1]. In GRAPHITE, a polyhedral model of the program is extracted directly from the GIMPLE Static Single Assignment (GIMPLE SSA) intermediate representation. Graphite transformations can be performed speculatively within the GRAPHITE's framework, and their result is output as GIMPLE SSA structures. Furthermore, the polyhedral model abstracts the control structures of the program and supports a powerful representation of dependence relations between computations, and ultimately, the concurrency available in the program.

The Data-Flow execution model, as opposed to the Control-Flow model underlying traditional imperative programming, does not rely on a program counter for execution, but rather on the availability of the input data for each operation. The Data-

1 The project is supported by the Research Promotion Foundation of Cyprus (Project: ΤΠΕ/ΠΛΗΡΟ/0308(ΒΕ)/06)

Driven Multithreading (DDM) model is a non-blocking multithreading execution model in which threads are scheduled for execution once all their input data is available. The Data-Driven Multithreading Virtual Machine (DDM-VM) is a virtual machine that supports DDM execution on homogeneous and heterogeneous multi-core systems. DDM-VM virtualizes the parallel resources of the underlying machine and uses a general, unified interface for writing DDM programs. The expressive power and the underlying program representation of GRAPHITE are a perfect match to the data dependence-driven programming model of DDM.

This paper describes our work on automatic DDM-VM code generation using the polyhedral model. Section 2 gives an overview of the DDM execution model and the DDM-VM implementation. Section 3 summarizes the key features of GRAPHITE. In section 4 we describe the automatic DDM-VM code generation method based on GRAPHITE. Section 5 presents a preliminary performance evaluation and finally Section 6 summarizes current project status and gives directions for future work.

2 The Data-Driven Multithreading Virtual Machine (DDM-VM)

The Data-Flow model [2][3][4] is a formal model that handles concurrency in a distributed manner and tolerates memory and synchronization latencies efficiently, thus effectively addressing two of the main challenges of the *Concurrency Era*. The Data-Driven Multithreading (DDM) model is an execution model that combines the Data-Flow model ability to exploit concurrency with the efficient execution of the Control-Flow model. Threads are scheduled based on data availability, however execution within a thread proceeds in a control-flow manner. Since DDM exploits course-grain parallelism, it also diminishes pipeline stalls and flushes. DDM decouples the execution from the synchronization part of the program and allows them to execute asynchronously. The core of the DDM model is the Thread Scheduling Unit (TSU) [5], which is responsible for the scheduling of threads at run-time based on data-availability. As the input data of a DDM thread is identified, it can be pre-fetched [6] before the firing of the thread reducing cache misses considerably.

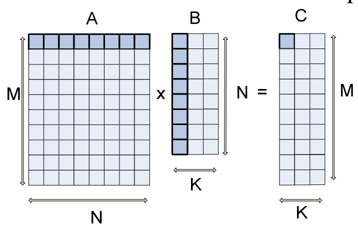
The Data-Driven Multithreading Virtual Machine (DDM-VM) [7] supports DDM execution on homogeneous and heterogeneous multi-core systems. It uses a general and unified representation of DDM programs based on C macros. Each DDM-VM program consists of two parts: the code of the DDM threads and the *dependency graph* describing the consumer-producer dependencies among the threads. At runtime the dependency graph is loaded into the TSU and used for the scheduling of the threads based on data-availability. Each thread in the graph is assigned a value equal to the number of its producer threads called the *Ready Count* (RC). When a thread finishes execution, the TSU uses the graph to identify and update the RC of its consumers. A thread is ready for execution when its Ready Count reaches zero and it is scheduled for execution once a core is available.

Currently, there are two implementations of the DDM-VM. The DDM-VM$_s$ for homogeneous multi-cores and the DDM-VM$_c$ for heterogeneous multicores, which has the Cell B.E. [8] as its main target. In the DDM-VM$_c$, the TSU is implemented as a software module running on the general purpose PPE core, while the execution of the DDM threads takes place on the SIMD SPE cores. A special module inside the TSU manages the data management between the Cell memory hierarchy implicitly. In the DDM-VM$_s$, the TSU is also implemented as a software module running on one of the cores, leaving the threads to execute on the other cores.

Hand-coded DDM-VM programs are represented using a set of C macros [7] that expand into calls to the DDM-VM runtime. We explain programming with the DDM-VM macros using the matrix multiplication example shown in Fig. 1. One possible representation of this example program in DDM is using three threads. The dynamic dependency graph of the DDM threads is illustrated in Fig. 2. Each invocation of a DDM thread is labeled with its corresponding <*context*>, which is a unique value distinguishing multiple invocations of the same thread. In the case of threads representing loops the *context* value corresponds to the loop indices.

A DDM-VM program consists of two parts. The first is the *main* function that is responsible for the initialization, creation of the dependency graph, starting the execution of the graph and post-execution. This part of the program is depicted in Fig. 3. The second part is the code of the DDM threads depicted in Fig. 4. The main part can also include one optional section shown in Fig. 5.

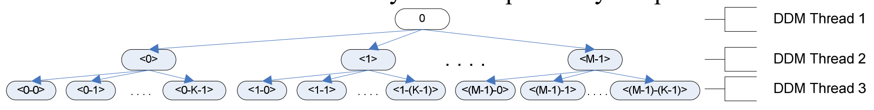
Semi-Transposed Matrix Multiplication



```
float A[M][N];
float B[K][N]; //this is B^T
float C[M][K];
int main(){
/*Data Initialization*/
for (m=0;i<M;m++)
 for (k=0;i<K;k++)
  multAdd(C[m][k],A[m][],B[k][]);
}/*C[m][k]+=A[m][1..N]*B[k][1..N]
*/
```

Fig. 1: The Matrix Multiplication Example

DDM Threads Dynamic Dependency Graph



threads are identified by their *context* value <context>

Fig. 2: Dynamic Dependencies across the DDM threads invocations

The **`DVM_SET_THREAD_TEMPLATE`** macro is used in the *main* function to load the *synchronization template* for each DDM thread into the TSU. Each thread is given a unique *Thread Identifier* (`Thread_Id`) and assigned its *readycount* value, the list of its consumer threads, the number of input data, the scheduling policy to use with the thread and the *arity* of the thread which indicates the nesting level in the case of threads representing loops. The templates of all the threads form the *dependency graph*. Once the loading of the graph is done, the **`DVM_UPDATE_THREAD`** macro is used to inform the TSU to decrement the *readycount* of invocation zero of `THREAD_1` from one to zero, making it ready for execution. The **`DVM_EXECUTE()`** macro is then used to start the execution of the TSU and the scheduling of threads to the cores. This call returns when all the threads in the graph finish execution.

```
int threads_main(int core_id)
{

INIT_RUNTIME(core_id);

DVM_THREAD_START(THREAD_1);

for(m=0; m < M; m++)
  DVM_UPDATE(CONS1,
    OP_SET_CONTEXT,m);

DVM_THREAD_END();
DVM_THREAD_START(THREAD_2);
GET_CONTEXT_S(DVM_CONTEXT,m);

for(k=0; k < K; k++)
  DVM_UPDATE(CONS1,
    OP_SET_CONTEXT,
    MAKE_CONTEXT_D(m,k));

DVM_THREAD_END();

DVM_THREAD_START(THREAD_3);
GET_CONTEXT_D(DVM_CONTEXT,
   m,k);

matmul(C[m][k],A[m][],B[k][]);

DVM_THREAD_END();
```

Fig. 3: The code of the DDM Threads (the *worker* function).

```
int main(int argc, char **argv)
{

// Data Initialization goes here

INIT_DDMSYSTEM();

DVM_SET_THREAD_TEMPLATE(THREAD_1,   //ThreadID=1/IFP
THREAD_2, 0                         //Consumers
1,                                  //ReadyCount=1
0,                                  //DFPNum=0
DVM_STATIC, 0,                      //Scheduling Mode
DVM_ARITY_0,                        //NestingLevel=0
SM_DEFAULT, 0);                     //DefaultSM
DVM_SET_THREAD_TEMPLATE(THREAD_2,   //ThreadID=2/IFP
THREAD_3, 0                         //Consumers
1,                                  //ReadyCount=1
0,                                  //DFPNum=0
DVM_MODULAR,MASK_CONTEXT,           //Scheduling Mode
DVM_ARITY_1,                        //NestingLevel=0
SM_DEFAULT, 0);                     //DefaultSM
DVM_SET_THREAD_TEMPLATE(THREAD_3,   //ThreadID=3/IFP
0, 0                                //Consumers
1,                                  //ReadyCount=1
3,                                  //DFPNum=3
DVM_MODULAR, MASK_CNTX,             //Scheduling Mode
DVM_ARITY_2,                        //NestingLevel=0
SM_DEFAULT, 0);                     //DefaultSM

//Update the ready count for THREAD_1, becomes=0
DVM_UPDATE_THREAD(THREAD_1,0);

//Start the execution of the TSU
DVM_EXECUTE();
//This call returns when all the threads in the
//program finish execution

//When all done shut down the system.
SHUTDOWN_DDMSYSTEM();
}
```

Fig. 4: The `main` function of the DDM program.

```
// DFPL definitions for enabling prefetching
DVM_CACHEFLOW_DFPL_START();//start of function called by TSU
int i,j;
DVM_START_DFPL(THREAD_3); // start of DFPL definition
GET_CONTEXT_D(DVM_CONTEXT,m,k);
DVM_SET_DFP(A[m],N*sizeof(int),DATA_IN);
DVM_SET_DFP(B[k],N*sizeof(int),DATA_IN);
DVM_SET_DFP(&C[m][k],sizeof(int),DATA_IN|DATA_OUT);
DVM_END_DFPL(); // end of DFPL definition
DVM_CACHEFLOW_DFPL_END(); // end of function called by TSU
```

Fig.5: The DFPL macros encoding the information of the data of the threads.

The first and second threads implement the two loops and the third thread implements the body of the inner loop, making a call to the `matmul()` function performing the row-column multiplication. The macros **`DVM_THREAD_START`** and **`DVM_THREAD_END`** mark the boundaries of each thread. The **`DVM_CONTEXT`** is a variable set by the runtime to the current value of the *context*. The **`GET_CONTEXT`** macro is used to retrieve the *context* components corresponding to the values of the loop indices (*m* and *k* in this example).

Upon a thread termination, the **`DVM_UPDATE`** macro informs the TSU to decrement the *readycount* of the thread consumers. It specifies the dynamic invocation of the consumer to update by designating the *context* value of that instantiation using the **`MAKE_CONTEXT`** macro. The *for* loop in the body of **`THREAD_1`** invokes the **`DVM_UPDATE`** macro for each value of *m,* which causes the TSU to decrement the *readycount* of the invocations of **`THREAD_2`** with a *context* value equal to *m,* making them ready for execution. Similarly, each invocation of **`THREAD_2`** invokes the **`DVM_UPDATE`** macro for each value of *k,* which eventually results in making invocations of `THREAD_3` with a *context* value equal to <m,k> ready for execution. Each such invocation runs in parallel and calculates the multiplication of row *m* of A with column *k* of B (actually row *m* of A with row *k* of B since it's transposed).

The information of the data of each thread is encoded using the macro **`DVM_SET_DFP`** as shown in Fig. 5. The address of the data for each invocation of a thread is defined in terms of the *context*. In the case of the DDM-VM*s* this information can be used by the TSU for prefetching. In the case of the DDM-VM*c* this information is necessary for the automatic management of the Cell memory hierarchy.

The default scheduling policy utilized by the TSU is to dynamically assign threads to cores in a way that maximizes load-balancing. This can be selected by passing DVM_DYNAMIC as the 6th parameter of the **`DVM_SET_THREAD_TEMPLATE`** macro. The programmer can also select a STATIC policy that schedules all the invocations of a thread to a certain core, a MODULAR policy that assigns the invocations using part of the *context modulo* the available cores, a Round Robin policy or a CUSTOM scheduling policy. For example, the scheduling policy of `THREAD_3` assigns the invocations of that thread in a modular fashion where invocations multiplying the same row are assigned to the same core.

3 The GRAPHITE Project

GRAPHITE (GIMPLE Represented as Polyhedra with Interchangeable Envelopes) [9][10][1] is a project that aims at adding static high level loop nest optimizations in GCC, based on the polyhedral representation.

The main task of the GRAPHITE pass in GCC is to extract the polyhedral model representation out of the GCC three-address GIMPLE representation, perform the various analyses and optimizations on the polyhedral model, and generate back the new GIMPLE three-address code that corresponds to the result of the transformations. The key steps in the GRAPHITE pass are as follows:

1. The Static Control Parts (SCoPs) are outlined from the control flow graph [10].
2. The polyhedral representation is constructed for each SCoP in a process called GPOLY construction.

3. Data Dependence analysis and transformations are performed, possibly multiple times. Transformations can be rolled back if unsuccessful.
4. Generation of GIMPLE code corresponding to transformed polyhedral model using CLooG.

SCoPs are Single-Entry-Single-Exit (SESE) regions of the control flow graph consisting of possibly non-perfectly nested/non-rectangular loops and conditionals with boolean expressions on affine inequalities. SCoP inputs are either scalar values (parameters) that are invariant in SCoP or array values which can be modified. SCoP outputs are arrays which have been modified in the SCoP and are used after the SCoP. SCoPs can contain calls to functions that are pure, but no calls to functions with side effects. The only memory references that are allowed are array accesses with affine subscript functions. The SSA and scalar evolution analysis framework of GCC are used in order to perform SCoP outlining.

GPOLY is the polyhedral information attached to each basic block in a SCoP and consists of iteration domain, schedule and data access functions. All these information is represented as systems of affine equalities and inequalities which can be manipulated using a polyhedral library like PPL (Parma Polyhedral Library). Initial scheduling functions for each basic block are deduced from the Loop Statement Tree (LST) showing the relative ordering of the basic blocks, and initial nesting structures for loops.

GRAPHITE uses an iterative and speculative transformation process. The application of a transformation is followed by a check of its legality and profitability. Currently, GRAPHITE implements loop interchange, loop strip-mining, loop distribution and loop-blocking. Loop blocking is a combination of strip-mining on all indices followed by loop interchange on all pairs of resulting indices, and succeeds only if the interchange reduces the amount of data accesses wrt. unmodified code.

The final stage in the GRAPHITE pass is GIMPLE code reconstruction using CLooG (Chunky Loop Generator). The iteration domains represented as multidimensional regions – possibly with guard conditions – are converted into loops and conditionals, and data references are converted into actual scalar and array accesses with all necessary address computations.

4 Implementing DDM support in GCC

The key to DDM code generation is the identification and explicit representation of DDM threads, DDM thread invocations (activations of DDM threads each with a specific context value), and dependences between individual DDM thread invocations. Since the GRAPHITE framework operates directly on sets of computations identified by values of loop indices, it provides a natural solution for deriving DDM threads and data dependences occurring within a SCoP region. Therefore, for the initial version of the DDM compiler it was decided to use the GRAPHITE framework as the core analysis and transformation engine, and leverage the GOMP and Autopar solutions as much as possible. This makes it possible to benefit from powerful and already validated compiler features while reducing the overall development effort and placing the focus on the DDM-specific issues.

While the core of the GRAPHITE analysis and transformation engine remained virtually unchanged, a new GRAPHITE-to-GIMPLE translation scheme was needed in order to put the reconstructed SCoP in a DDM-compliant form. Additionally, since the DDM runtime system can exploit parallel loops present at any depth in a loop tree, it was necessary to collect and record the properties of the generated loops until the very last stages of the GRAPHITE-to-GIMPLE translation process.

In the prototype implementation we focused on the functional correctness of the generation process, leaving performance tuning considerations for a later project phase. For this reason, the major part of the development efforts was centered on the code generation path. GRAPHITE loop analysis and transformation mechanisms were used unmodified, in particular wrt. controlling loop blocking and identifying parallel loops. The modifications were concentrated primarily in the GRAPHITE-to-GIMPLE part (clast_to_gimple), but several other modifications were also needed in the function outlining mechanism.

In the DDM code generation flow, the translation of the GRAPHITE representation of a SCoP (a CLAST, or CLooG Abstract Syntax Tree) into the GIMPLE-SSA code is split into four major stages:

1. Reconstruction of imperative control structures and recording detailed information about each loop created.
2. Identification of the loops which should be converted into DDM threads.
3. If qualifying loops are found, outlining of the outermost loop(s) to a worker function and insertion of the corresponding DDM API calls around the outlining point(s).
4. Refinement of the outlined function to expose further threads.

The reconstruction of imperative control structures takes care of recording detailed information about the loops being constructed. During the reconstruction of each loop, the CLAST-to-GIMPLE code checks whether in the context of the already reconstructed code the loop carries any dependences and if not, marks the loop as parallel. To support DDM code generation we extended the reconstruction code to record additional information needed to convert a loop nest into a DDM thread body and to issue the necessary *context* manipulation and thread activation primitives.

Recall that the DDM runtime scheme requires that the DDM threads are combined into a single worker function. This means that if a CLAST is selected for conversion into a set of DDM threads, it must be outlined as a whole into a function. If specific inner loops in the outlined CLAST need to be converted into additional DDM threads (e.g., in order to constrain the maximum number of active threads), the refinement must take place inside the outlined function.

The identification of the largest subset of control structures to be converted into DDM threads is a task which requires the most complex decisions. The selection criteria for this phase depend directly on the properties of target architecture such as the amount of per-node storage available, the sizes of the TSU queues, the maximum/recommended number of simultaneously active DDM nodes etc. Therefore, in order to reduce development complexity, this stage is currently controlled manually from the command line of the compiler.

The outlining stage relies on the single-entry, single-exit (SESE) region outlining mechanism initially developed for OpenMP [11]. However, it was modified to account for two specificities of DDM: an implicit parameter passing scheme based on Data Frame Pointers, and the incremental construction of the outlined function.

In the OpenMP parameter-passing approach, the worker function is given an explicit pointer-to-void call argument, which is interpreted as a pointer to the data frame used to communicate between the caller and the worker function. In the current implementation of DDM there is no such argument, and the DDM Data Frame Pointer (DFP) had to become a global variable visible by both the caller of the DDM code and by the DDM worker function.

The construction of the DFP structure directly reuses the mechanism for exposing nonlocal data access developed for Autopar (cf. function separate_decls_in_region). The DFP computed in this way contains pointers to arrays and integer fields corresponding to outer loop iterators referenced inside the outlined CLAST.

The last stage of the translation is to refine each outlined loop tree by exposing a new thread for each loop which should be converted to a DDM thread. In the current implementation, all such threads share a single DFP corresponding to all non-local variables accessed from the outlined CLAST.

The DDM worker function consists of an initialization preamble, and a potentially infinite loop which repeatedly requests a new thread description from the TSU and processes the thread invocation defined by the thread identifier and *context* returned by the **`GetNextThread()`** call:

```
int threads_main (int coreid)
{
      unsigned int raw_context;
      unsigned int thread_id;
      INIT_RUNTIME(coreid);

      while (1) {
         Check_Fin (coreid);//ensures the TSU is notified of termination
         GetNextThread (&thread_id, &raw_context, coreid);
              switch (thread_id) {
                      case THREAD_1 :
                              // decode context
                              // execute thread body
                              break;
                      case THREAD_2 :
                              // decode context
                              // execute thread body
                              break;
                      ...
                      default:
                              return thread_id;
              }
      }
}
```
Fig. 6: The structure of the worker function

The TSU request is blocking; once the worker function is unblocked, it enters a switch statement driven by the DDM thread identifier supplied by the TSU. The selected branch of the switch statement decodes the raw *context* supplied by the TSU and executes the corresponding thread body with the decoded *context* values. If the TSU supplies an invalid thread identifier, the worker function returns.

All corresponding thread *template* load calls are placed at the original site of the first outlining, such that the loading of the dependency graph is completed upon calling the function starting the execution of the graph (`DDM_Run()`).

A loop marked as parallel during the GIMPLE reconstruction pass is transformed into a thread whose instances are indexed by a context derived from the loop index. The body of the thread is the body of the loop, with index increment and looback test removed. All instances are activated in parallel with context (or context field) value ranging from the minimal to the maximal value of the loop index in strides of 1 by issuing a multiple-thread immediate update call at the end of the basic block which precedes the original location of the loop in the control flow.

A loop which was not marked as parallel during the GIMPLE reconstruction pass carries at least one dependence and is transformed to a sequential (self-firing) thread. The thread instances are indexed by a context (or context field) whose initial value is the lower bound of the loop. The body of the thread is the body of the loop, modified to perform conditional TSU update operations instead of loopback and loop exit jumps. If the incremented value of the loop index (i.e., context or context field value) is less-or-equal than the maximum value of the loop index, the thread re-fires itself with the incremented value of the context (resp. context field). If the incremented value exceeds the upper bound, the TSU update operation is issued with additional flag to indicate that no further threads will be activated. The call which activates the first instance of the thread (corresponding to the lower bound of the loop) is issued at the location immediately preceding the converted loop in the control flow. For the outermost outlined loop, it will be the basic block preceding the origin of outlining.

The above translation rules are sufficient if the loop is the outermost outlined loop of a loop-and-statement tree and contains no further threads: the ordering of the outlined loop and the code that precedes (resp. follows) it will be ensured by the blocking behavior of the `DDM_Run()` function. If multiple loops will be converted to threads in the outlined code, additional precautions must be taken in order to preserve program semantics.

Given that GRAPHITE does not (yet) provide do-across transformations, the DDM code generation process must enforce the serialization of iterations containing dependent statements.

A conservative, barrier-like approach is to ensure that any parallel code executing inside a sequential loop has terminated before the next iteration of the sequential loop can start. This constraint can be relaxed in the case of sequences of statements if it can be demonstrated at compile time that there is no dependence between statements (and/or statement sequences) S_1 and S_2 located at the same nesting level.

For a loop L (whether parallel or sequential) which 1) is not the outermost loop of an outlined region and 2) within the outlined region is contained in a sequence of statements or an outer sequential loop, it is sufficient to ensure that the sequential ordering around the current loop is preserved, i.e., that

1. The execution of L does not begin (the threads of L do not enter the ready-to-run state) until all preceding threads and statements of the current iteration of the enclosing loop have completed.
2. The execution does not proceed beyond the first sequence point following loop L until all threads of L have finished executing.

The above barrier-based approach to thread ordering is over-constraining the execution order, unnecessarily reducing the potential of the program. In order to expose more parallelism it is necessary to take into account individual loop-carried and inter-loop dependences, and the dependence distances between sources and sinks of each dependence. However, contrary to a pure dataflow approach where each statement or operator can be individually triggered by data availability, the DDM execution scheme requires a coarser granularity of firings.

The determination of the appropriate firing granularity of threads interacts closely with the size-related thread granularity constraints: a coarser granularity of firing reduces DDM runtime overheads, but it also reduces the available concurrency. On the other hand, threads may have to be refined in order to meet storage constraints of the individual compute nodes. This is why the initial implementation of GCC support for DDM relied on a coarse-grain, single loop level thread extraction.

In future development stages, the constraints of firing granularity and storage requirements will be combined into a single, multi-level strategy of thread forming and inter-thread dependence management. In particular, we plan to leverage precise data dependence information from GRAPHITE (esp. affine dependence functions) to implement do-across dependences between dependent loop iterations so that the effective degree of concurrency of the code is maximized without incurring excessive overheads from the DDM runtime system.

5 Preliminary Performance Evaluation

While still developing the functionality of the GCC-based DDM compilation flow, we carried out an initial evaluation of the performance of the generated code. In order to isolate the impact of the code generation scheme and to limit the possible bias introduced by differences in data layouts, we used a semi-transposed matrix multiplication, which is very suitable for both sequential optimization and Graphite-driven parallelization. The algorithm performs row-major access on both the direct matrix A and the transposed matrix B, which enables aggressive sequential optimizations and makes GRAPHITE's loop blocking transform succeed.

We compared the sequential code generated by GCC (v4.6, SVN revision 163224) at optimization levels -O0, -O1 and -O2 with the parallel code in which the outermost block index was used as the DDM thread index, using all combinations of block tile sizes of 32, 64, 128 and 256 elements and optimization levels -O1 and -O2 for a 2K x 2K single-precision floating-point matrix multiply. The experiments were run on a dual-Xeon E5320 server (2 CPUs, 8 cores, 1.86 GHz, 2 x 8MB cache). The reference is the sequential code compiled at optimization level -O2. The results are summarized in Table 1 below.

Mode	-O0		-O1		-O2	
	duration (s)	speedup	duration (s)	speedup	duration (s)	speedup
sequential	68.856	0.24	16.489	1.01	16.586	1.00
DDM - 32	N/A	N/A	7.636	2.17	6.961	2.38
DDM - 64	N/A	N/A	6.820	2.43	6.823	2.43
DDM - 128	N/A	N/A	7.110	2.33	7.238	2.29
DDM - 256	N/A	N/A	8.066	2.06	8.406	1.97

Table 1: Performance of DDM matrix multiplication generated using GCC

These early results indicate that even without tuning and in the presence of overheads caused by redundant control structures, the generated DDM code outperforms the sequential implementation by a factor 2.4x. Also, the tile size has a non-linear influence on performance, yielding best result at tile size 64. Significantly better figures should be achievable by tuning the tiling factor (including the possibility of making it dependent on the specific indices) and by removing the overheads introduced by GRAPHITE's loop blocking transform, which performs strip mining on all indices and leaves the additional indices in, even if not used for blocking. The latter will benefit from introducing the linearization transform, which will also be key to more general data layout reorganizations such as those applied when hand-optimizing DDM programs.

6 Perspectives and Future work

In the process of designing and implementing the DDM support in GCC we were able to leverage many existing features of the compiler: SESE region outlining, Autopar's variable analysis and most importantly, the GRAPHITE framework. While we had to adapt or extend the existing mechanisms to accommodate the requirements of DDM, the modifications to the existing mechanisms are very limited and correspond to overcoming certain implicit assumptions, as in the case of single-step outlining of SESE regions.

The immediate evolutions of the current implementation of the DDM compilation flow include both performance-related and functionality-related aspects. Functionality extensions of the current DDM support in GCC are twofold. Adding the support for heterogeneous multiprocessors such as the Cell BE will require a careful implementation of data management (prefetching, locality optimizations, etc.) and a possible revision of the code generation approach. It will also result in automating the selection of thread granularity driven by both data and code size limitations.

Based on the infrastructure developed during this first phase of the DDM compiler support implementation, we plan to extend the features of the current code generator for both symmetrical and heterogeneous multiprocessor architectures to further automate the selection of thread candidates and choice of transformation parameters.

Another area of research is the interaction between GRAPHITE and vectorization. Currently, vectorized code is rejected by GRAPHITE, while an appropriate representation of the vectorization constraints and transformations could in principle be used as GRAPHITE input, either to preserve already vectorized code, or to guide a vectorization post-pass. Similarly, causes of GRAPHITE failures to perform a transformation should be better reported to the programmer by providing human-readable feedback from the GRAPHITE engine [12]. Finally, inter-SCoP dependence analysis would extend the current scope of GRAPHITE analyses and transformations to whole functions, yielding more potential for thread detection and creation.

References

1. Konrad Trifunovic, Albert Cohen, David Edelsohn, Li Feng, Tobias Grosser, Harsha Jagasia, Razya Ladelsky, Sebastian Pop, Jan Sjodin, and Ramakrishna Upadrasta, GRAPHITE Two Years After First Lessons Learned From Real-World Polyhedral Compilation. GROW'10, Pisa, Italy, Jan. 2010.
2. J. B. Dennis, "First Version of a Data Flow Procedure Language," in Programming Symposium, Proceedings Colloque sur la Programmation. London, UK: Springer-Verlag, 1974, pp. 362–376.
3. Arvind and K. P. Gostelow, "The U-Interpreter," IEEE Computer, vol. 15, no. 2, pp. 42–49, 1982.
4. I. Watson and J. Gurd, "A Practical Data Flow Computer," IEEE Computer, vol. 15, no. 2, pp. 51–57, 1982.
5. P. Evripidou, "Thread Synchronization Unit (TSU): A Building Block for High Performance Computers," In: Proceedings of the International Symposium on High Perfomance Computing, Fukuoka, Japan., 1997.
6. C. Kyriacou, P. Evripidou, and P. Trancoso, "Cacheflow: A Short-Term Optimal Cache Management Policy for Data Driven Multithreading," Proc. EuroPar-04, pp. 561–570, Aug. 2004.
7. S. Arandi, P. Evripidou, "Programming multi-core architectures using Data-Flow techniques", IC-SAMOS 2010: 152-161
8. J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer, and D. Shippy. 2005. Introduction to the Cell multiprocessor. IBM J. Res. Dev. 49, 4/5 (July 2005), 589-604.
9. Sebastian Pop, Albert Cohen, Cedric Bastoul, Sylvain Girbal, Georges-Andre Silber, Nicolas Vasilache, GRAPHITE: Polyhedral Analyses and Optimizations for GCC. GCC Summit 2006, Ottawa, Canada.
10. C. Bastoul and A. Cohen and S. Girbal and S. Sharma and O. Temam. Putting Polyhedral Loop Transformations to Work, LCPC'16 International Workshop on Languages and Compilers for Parallel Computers, LNCS 2958, pages 209–225, College Station, Texas, October 2003.
11. D. Novillo, OpenMP and automatic parallelization in GCC, GCC Developers' Summit, Ottawa, Canada, June 2006.
12. Per Larsen, Razya Ladelsky, Sven Karlsson, Ayal Zaks. Compiler-Driven Code Comments and Refactoring. 4th Workshop on Programmability Issues for Multi-Core Computers (MULTIPROG-11), Heraklion, Greece, January 2011.