Polyhedral Compilation Support for C++ Features: A Case Study with CPPTRAJ

Amit Roy[1], Daniel Roe[2], Mary Hall[1](✉), and Thomas Cheatham[2]

[1] School of Computing, University of Utah, Salt Lake City, UT 84112, USA
mhall@cs.utah.edu

[2] Department of Medicinal Chemistry, University of Utah, Salt Lake City, UT 84112, USA

Abstract. This paper reveals challenges in migrating C++ codes to GPUs using polyhedral compiler technology. We point to instances where reasoning about C++ constructs in a polyhedral model is feasible. We describe a case study using CPPTRAJ, an analysis code for molecular dynamics trajectory data. An initial experiment applied the CUDA-CHiLL compiler to key computations in CPPTRAJ to migrate them to the GPUs of NCSA's Blue Waters supercomputer. We found three aspects of this code made program analysis difficult: (1) STL C++ vectors; (2) structures of vectors; and, (3) iterators over these structures. We show how we can rewrite the computation to affine form suitable for CUDA-CHiLL, and also describe how to support the original C++ code in a polyhedral framework. The result of this effort yielded speedups over serial ranging from 3× to 278× on the six optimized kernels, and up to 100× over serial and 10× speedup over OpenMP.

1 Introduction

CPPTRAJ is a biomolecular analysis code that examines results of simulations that are represented as time series of three-dimensional atomic positions (i.e., coordinate trajectories) [1]. CPPTRAJ is an MPI and OpenMP code distributed as part of the AmberTools suite, a widely-used set of tools for complete molecular dynamics simulations, with either explicit water or implicit solvent models [2], and is also available on GitHub [3]. Historically, the analysis function is less compute-intensive than the simulation, and less attention has been paid to its parallelization. As Amber simulations scale to larger supercomputing systems, it is desirable to perform analysis functions in situ during simulation to reduce data movement and storage. Thus, analysis has become a more significant component of simulation time, and worthy of renewed attention paid to its parallelization, especially in light of new architectures.

Parallelization within the `Action` class computations offered an unexploited opportunity for thread-level parallelism on GPUs. We adapted one of the more time-consuming analyses in CPPTRAJ, the `Action_Closest`, which determines the `N` closest solvent molecules to `M` solute atoms where `N` and `M` are both user-specified. This calculation can require millions of distance calculations for each

L. Rauchwerger (Ed.): LCPC 2017, LNCS 11403, pp. 26–35, 2019.
https://doi.org/10.1007/978-3-030-35225-7_3

trajectory frame, to use GPUs. To ease the programming challenges of migrating CPPTRAJ to use GPUs, we employed CUDA-CHiLL, which generates CUDA code from a sequential implementation [4,5]. CUDA-CHiLL is a lightweight GPU-specific layer for CHiLL, a source-to-source code translator that takes as input sequential loop nest computations written in C, performs transformations, and generates optimized sequential or parallel C. A separate input called a *transformation recipe* describes high-level code transformations to be applied to the code; this recipe can either be automatically generated [5] or specified by the programmer. The underlying compiler technology relies on a *polyhedral* abstraction of loop nest computations, where loop iteration spaces are represented as polyhedra.

CUDA-CHiLL has a C++ frontend, but has primarily been applied to C codes. We discovered that some of the C++ features are difficult to represent in a polyhedral framework: (1) structures of arrays; (2) C++ iterators; and, (3) a vector library. We initially modified the code so that CUDA-CHiLL could analyze it and generate GPU code. The resulting code achieves high performance, meeting the goals of the optimization exercise and providing a template to the CPPTRAJ team for further parallelization. We then considered how to extend CUDA-CHiLL to support these features. The contributions of this paper are: (1) a description of a successful parallelization of CPPTRAJ for GPUs; (2) analysis of barriers to automatic parallelization in CUDA-CHiLL; and, (3) extensions to polyhedral compiler technology to support the C++ features of this code.

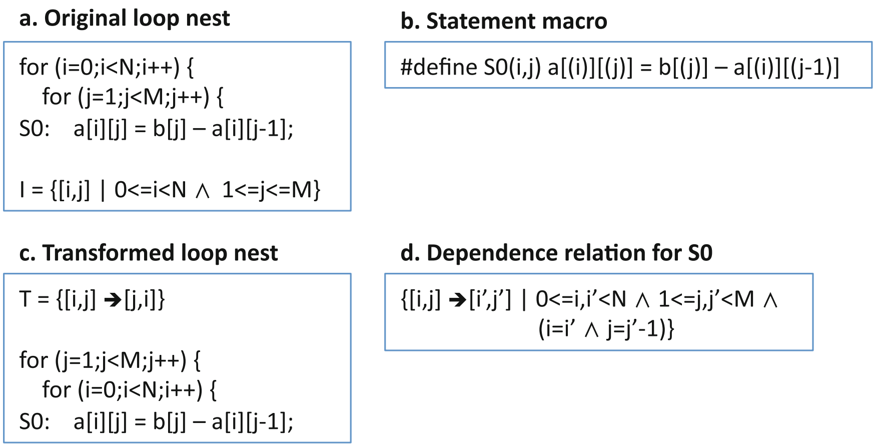
2 Background and Related Work

We describe the foundations of polyhedral transformation and code generation technology, and tease out key concepts in extending its support.

2.1 Polyhedral Compiler Frameworks

Polyhedral frameworks describe the iteration space for each statement in a loop nest as a set of lattice points of a polyhedron. Loop transformations can then be viewed as mapping functions that convert the original iteration space to a transformed iteration space, providing the compiler a powerful abstraction to transform a loop nest without being restricted to the original loop structure [6]. To verify correctness of iteration space remappings, the compiler employs *dependence analysis*, which detects possible accesses to the same memory location, where one of the accesses is a write. Reordering a statement's execution order is valid as long as it preserves all data dependences [7]. Once transformations are proven safe through dependence analysis, the code corresponding to the transformed iteration space may then be generated by polyhedra scanning [8–12].

Let us consider for example, the loop permutation transformation applied to the loop nest in Fig. 1(a), with the iteration space `I` represented as an integer tuple set. The original statement is replaced by a statement macro as shown in Fig. 1(b). The loop permutation transformation `T` in Fig. 1(c), which permutes



a. Original loop nest

```
for (i=0;i<N;i++) {
   for (j=1;j<M;j++) {
S0:   a[i][j] = b[j] – a[i][j-1];

I = {[i,j] | 0<=i<N ∧  1<=j<=M}
```

b. Statement macro

```
#define S0(i,j) a[(i)][(j)] = b[(j)] – a[(i)][(j-1)]
```

c. Transformed loop nest

```
T = {[i,j] ➔[j,i]}

for (j=1;j<M;j++) {
   for (i=0;i<N;i++) {
S0:   a[i][j] = b[j] – a[i][j-1];
```

d. Dependence relation for S0

```
{[i,j] ➔[i',j'] | 0<=i,i'<N ∧ 1<=j,j'<M ∧
                  (i=i' ∧ j=j'-1)}
```

Fig. 1. An example of a loop permutation transformation.

the order of the loops, takes `I` as input and returns an output integer tuple. The code generator then employs polyhedra scanning of the resulting iteration space to generate the output code shown. To determine safety of the transformation, dependence relations are extracted from examining the iteration space and array accesses, as in Fig. 1(d). In this case, while there is a dependence between reads and writes of `a`, permutation is safe because it does not reverse the dependence on `a`. The statement is not specified in the set representation, and therefore the loop body contains statement macros. The transformed loop need only pass to the statement macro the original iterators for the statement as functions of the new loop iterators.

2.2 Support for C++ Code

Many polyhedral frameworks are embedded into C and C++ compilers and leverage parsing of C++ code into an abstract syntax tree (e.g., PolyOpt, PSSC [13], Polly [14]). Some polyhedral compilers generate CUDA code as in this work [15,16]. Such compilers typically look for analyzable regions of code amenable to polyhedral optimization, called Static Control Parts (SCoPs) such that all loop bounds and conditionals are affine functions of enclosing loops. Certain C++ code constructs may appear to be non-affine to a polyhedral compiler, and therefore these portions of the code would be ignored and not optimized, even though they could be rewritten into an affine form. Notably, analysis and transformation merely needs to extract dependence relations and statement macros as functions of loop indices. *We consider in this paper such examples whereby we reason about C++ code and represent the code in statement macros, extract iteration spaces to facilitate transformation and code generation, and extract dependence relations to determine safety of transformations.*

3 Code Modifications and Extensions for CPPTRAJ

This section highlights the C++ features that we modified to pass the CPPTRAJ code through CUDA-CHiLL, and discusses possible extensions.

```
 1 void Action_Closest::Action_NoImage_Center(Frame&, double maxD)
 2 {
 3   double Dist;
 4   int smol;
 5   std::vector<int>::const_iterator satom;
 6 
 7   Vec3 maskCenter = frmIn.VGeometricCenter( distanceMask_ );
 8   for (smol=0; smol < Nsmols_; smol++) {
 9       SolventMols_[smol].D = maxD;
10       for (satom = SolventMols_[smol].solventAtoms.begin();
11            satom != SolventMols_[smol].solventAtoms.end();
12            ++satom)
13     {
14 
15         double *a1 = maskCenter.Dptr(); //center of solute molecule
16         double *a2 = frmIn.XYZ(*satom);
17 
18         double x = a1[0] - a2[0];
19         double y = a1[1] - a2[1];
20         double z = a1[2] - a2[2];
21 
22         Dist = (x*x + y*y + z*z);
23 
24         if (Dist < SolventMols_[smol].D)
25           SolventMols_[smol].D = Dist;
26     }
27   }
28 }
29 \vspace*{-.1in}
```

Listing 1.1. Original code for `Action_Closest`.

3.1 Changes Irrelevant to a Polyhedral Framework

The original C++ code is shown in Listing 1.1. A few constructs not supported by CUDA-CHiLL are not fundamental, and extensions to the implementation are straightforward. The required changes, which will not be discussed further, include (1) use of member functions of a class, and reference to member fields, which should be replaced with C functions and parameters; (2) control flow simplifications that would benefit from more sophisticated data-flow analysis; and, (3) the *min* calculation over `Dist`, which should be recognized as a reduction.

3.2 Other Ways of Expressing Loops over Arrays in C++

Additional required changes show C++ constructs that are comparable to standard loop nests over dense arrays, but are expressed differently from C. The reference in line 15 to `maskCenter` returns a variable of type `Vec3`, which is a simple datatype for representing 3D coordinates. The reference in line 16 to `frmIn.XYZ` returns a pointer to the position inside the Frame datatype's internal 3D coordinate array corresponding to atom `n`. Since these are read-only variables, it is sufficient to ignore the references since they cannot carry a dependence. However, in the more general case where they may also be written, it is useful to recognize that these types actually represent an array of three doubles.

The second kind of vector represented by `SolventMols_` adds more complexity to the analysis. It is declared as `std::vector⟨MolDist⟩`. That is, it uses the vector data type from the C++ standard template library. The code loops over the elements of this vector using a C++ iterator, `satom`.

A key observation is that these are implemented similarly to unit-stride access to arrays, but the compiler must be extended to recognize this. For our experiments, we have made these changes explicit. Referring back to Sect. 2.1, it is realistic to support these because we only need to extract three things from the code: (1) the iteration space of the loop nest; (2) the statement macro; and, (3) the dependence relations.

First, the loop nest needs to be rewritten in the code representation leading to an affine iteration space. The following rewrite is safe if you know that these vectors are stored contiguously in memory and the meaning of the `begin()`, `end()` and `size()` functions [17].

```
ub = SolventMols_[smol].solventAtoms.size();
I = {[smol,satom] | 0<=smol<Nsmols_  && 0<=satom<ub}
```

For the statement macros, we can leave line 15 as written in this case. But for line 16, we would like to rewrite so that if we are to modify the iteration space for the `satom` loop, we will be able to update the access in the context of the loop indices. The same is true for the reduction statement at lines 24 and 25. Therefore, the statement macros are as follows:

```
#define S16(smol,satom)
      double *a2 = SolventMols_[(smol)].solventAtoms[(satom)]
#define S24(smol,satom)
      SolventMols_[(smol)].D = min(Dist,SolventMols_[(smol)].D)
```

Finally, we consider the dependence relations arising from the statements that reference these vectors. As the statements at lines 15 and 16 are read-only accesses to the `maskCenter` and the data associated with the solvent atom, there are no dependence relations. For the access at lines 24 and 25, after the reduction transformation is performed as described above, the following dependence relation arises between read and write of `SolventMols_[smol].D`.

```
{[smol,satom]->[smol',satom'] | 0<=smol,smol'<NSmols_ &&
                                 0<satom,satom'<ub && smol=smol'}
```

This discussion assumes that the compiler can perform dependence analysis on fields in structures. This is a straightforward extension, where indexed fields are treated as arrays, and distinct fields are considered independent.

3.3 CUDA Code Generation and Application Integration

CUDA-CHiLL was applied to manually modified code to arrive at the output kernel code in Listing 1.2 and scaffolding code (not shown). The problem size is fixed to the sample input used for the experiments in Sect. 5. The generated code was derived using the CUDA-CHiLL script below.

```
__global__ void Action_No_image_GPU(double *D_,double *maskCenter,
    double (*SolventMols_)[965][3])
{
  int satom;
  int bx;
  int tx;

  double maxD;
  double Dist;
  double newVariable0;

  bx = blockIdx.x;
  tx = threadIdx.x;
  newVariable0 = D_[tx + 32 * bx];
  newVariable0 = maxD;

  for (satom = 0; satom <= 15021; satom += 1) {
    Dist = (pow(maskCenter[0] - SolventMols_[smol][satom][0],2) +
            pow(maskCenter[1] - SolventMols_[smol][satom][1],2) +
            pow(maskCenter[2] - SolventMols_[smol][satom][2],2));
    newVariable0 = (min(Dist,newVariable0));
  }
  D_[tx + 32 * bx] = newVariable0;
}
\vspace*{-.1in}
```

Listing 1.2. Kernel output of CUDA-CHiLL.

```
init("simple_action_noImage.c", "Action_NoImage_Center",0)
NA=15022
NM=965
TI=32
TJ=3*NM/TI
tile_by_index(0,{"smol"}, {TI}, {l1_control="ii"}, {"ii","smol"})
cudaize(0,"Action_No_image_GPU",
           {D_=NM*3, SolventMols_=NA*3,maskCenter=3},
           {block={"ii"}, thread={"smol"}},{})
copy_to_registers(0, "satom", "D_")
```

It is only safe to parallelize the outermost loop as the inner loop carries a dependence on `D_[smol]`. Therefore, this simple script creates two levels of parallelism for the outermost loop using the `tile_by_index` command. Each thread then computes one element of `D_`. To avoid unnecessary memory accesses, the `copy_to_registers` command is used to locally store `D_[smol]` in `newVariable0` during the majority of a thread's execution. The `cudaize` command marks the outermost two loops to serve as block and thread indices, whose sizes are controlled by `TI` and `TJ` derived from tuning. Note that different transformation recipes will lead to very different generated codes.

Five more member functions were also replaced with CUDA kernels. These all had similar structure and C++ features as compared to the code in Listing 1.1, but some had more computation at each point. We used the generated CUDA code as a template for the other kernels, and replaced the computation at the innermost loop. The CUDA code was then integrated back into the application with some additional functions calls from the `Action_Closest` class. We also inserted timing functions within a combined CUDA harness code for the kernels. Therefore, the impact in terms of coding changes on the application was not significant, but the performance gains were substantial, as shown in Sect. 5.

4 Incorporating Knowledge of Library or Class Properties

The previous section shows it is certainly feasible to represent the C++ constructs in this code as affine. However, the question arises as to how to embed knowledge into the compiler of the C++ STL or even a user class. For something as widely used as the STL, we could treat it as part of the C++ language and integrate these transformations into the CHiLL compiler directly. However, this approach would not apply to any user-defined class.

We propose to take advantage of CHiLL's existing *transformation recipe* interface to extend the compiler to convey this additional information. This concept of programmability of transformation recipes has been used before in adding CUDA support through a programming language interface [4], but in that case it was composing and reinterpeting existing CHiLL commands and modifying the output only. Here, we need a way of reinterpreting the input. We propose a new command in a transformation recipe called *scopInfo*:

```
scopInfo(loop, IS={affine_relation}, SM={statement_macros},D={deps})
```

This is one way to convey information to the compiler, before it attempts to analyze the code, that this analysis should permit extensions to whatever is already supported by CHiLL. This approach is similar to rewrite rules that are supported in domain-specific compiler frameworks such as DeLite [18], but specifically provides the inputs of a polyhedral framework to facilitate dependence analysis, iteration space reordering and code generation.

While such an extension could make it possible for a programmer to add *scopInfo* commands to their recipes, it may be too low-level for the average programmer. However, a custom preprocessing phase could be added to the framework to derive specialized information such as this in a domain-specific or library-specific way, particularly if the recipes are automatically generated as in [5]. We foresee such an extension would make it possible to convey other information to the compiler useful to loop nest optimization for HPC applications, such as for example, how to interpret user-defined domain decompositions.

5 Experimental Results

The GPU-enabled version of CPPTRAJ was then executed on the NCSA Bluewaters supercomputer, and compared against an MPI-only implementation and an MPI+OpenMP implementation. Bluewaters has two types of nodes, namely *XE* or *XK*. *XE* nodes have 2 AMD 6276 Interlagos processors while *XK* nodes have a single Interlagos processor and a GPU accelerator, an NVIDIA GK110 (K20X) Kepler GPU with 2688 CUDA cores. Both *XE* and *XK* nodes have 64 GBytes of memory. We used a molecular system for our experiment as a good proxy for typical real world usage, consisting of 4143 solute atoms and 15022 solvent molecules, resulting in up to 62M distance calculations for each frame.

As described in Sect. 3, the CPPTRAJ code was extended to replace six `Action` member functions with calls to CUDA kernels. The six kernels are divided into two groups: one group calculates distance with respect to the solvent molecule's center as represented by the code in Listing 1.2; the other calculates distance with respect to each atom contained within the solvent molecule. Figure 2(left) compares speedup over serial for each GPU kernel. Speedups range from 3× to 278×, with the *Non-center* kernels exhibiting a higher speedup. Each of the 3 kernels in each group is furthermore separated by the type of imaging method they use during the distance calculation. The labels *Ortho* and *Non-Ortho* refer to orthorhombic and nonorthorhombic, respectively, indicating the unit cell shape. Non-orthorhombic distance calculations are more compute-intensive as they check the "self" unit cell plus 26 images.

We now compare performance of the *Non-Center, Non-Ortho* kernel to the original OpenMP code within the full CPPTRAJ MPI code in Fig. 2(right). On a single node, the GPU version is rougly 10× faster than the OpenMP version. The substantial parallelism exhibits strong scaling as we deploy the application across multiple nodes, ranging from 1 to 32.

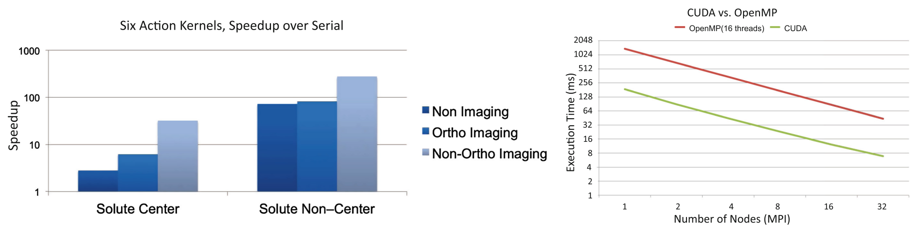


Fig. 2. Performance measurements on Blue Waters, showing speedup over serial of all Action kernels (left); OpenMP comparison and strong scaling within MPI code (right).

6 Conclusion

This paper has explored using polyhedral compiler technology to parallelize for GPUs key computations in CPPTRAJ, a real-world analysis code used for molecular dynamics trajectory data written in C++. The primary goal of this work was to derive high-performance GPU code for CPPTRAJ. At the same time, we explored the gaps in the CUDA-CHiLL framework for supporting C++ code and proposed how to extend polyhedral compiler technology to support C++ features, including the vectors in the standard template library.

We believe interactions such as this between HPC tool researchers and application developers on real applications lead to tools that better meet user needs while aiding applications in their migration to the variety of current and future architectures that require significant application changes.

Acknowledgment. This research is part of the Blue Waters sustained-petascale computing project, which is supported by the National Science Foundation (awards OCI-0725070 and ACI-1238993) and the state of Illinois. Blue Waters is a joint effort of the University of Illinois at Urbana-Champaign and its National Center for Supercomputing Applications.

References

1. Roe, D.R., Cheatham, T.E.: Ptraj and cpptraj: software for processing and analysis of molecular dynamics trajectory data. J. Chem. Theory Comput. **9**(7), 3084–3095 (2013). https://doi.org/10.1021/ct400341p. pMID: 2658398
2. http://ambermd.org
3. https://github.com/Amber-MD/cpptraj
4. Rudy, G., Khan, M.M., Hall, M., Chen, C., Chame, J.: A programming language interface to describe transformations and code generation. In: Cooper, K., Mellor-Crummey, J., Sarkar, V. (eds.) LCPC 2010. LNCS, vol. 6548, pp. 136–150. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-19595-2_10
5. Khan, M., Basu, P., Rudy, G., Hall, M., Chen, C., Chame, J.: A script-based autotuning compiler system to generate high-performance cuda code. ACM Trans. Archit. Code Optim. **9**(4), 31:1–31:25 (2013). https://doi.org/10.1145/2400682.2400690
6. Feautrier, P.: Automatic parallelization in the polytope model. In: Perrin, G.-R., Darte, A. (eds.) The Data Parallel Programming Model. LNCS, vol. 1132, pp. 79–103. Springer, Heidelberg (1996). https://doi.org/10.1007/3-540-61736-1_44
7. Allen, R., Kennedy, K.: Optimizing Compilers for Modern Architectures: A Dependence-Based Approach. Morgan Kaufmann Publishers, Burlington (2002)
8. Ancourt, C., Irigoin, F.: Scanning polyhedra with DO loops. In: Symposium on Principles and Practice of Parallel Programming, April 1991
9. Kelly, W.A.: Optimization within a unified transformation framework. Ph.D. dissertation, University of Maryland, December 1996
10. Quilleré, F., Rajopadhye, S.: Generation of efficient nested loops from polyhedra. Int. J. Parallel Program. **28**(5), 469–498 (2000)
11. Vasilache, N., Bastoul, C., Cohen, A.: Polyhedral code generation in the real world. In: Mycroft, A., Zeller, A. (eds.) CC 2006. LNCS, vol. 3923, pp. 185–201. Springer, Heidelberg (2006). https://doi.org/10.1007/11688839_16
12. Chen, C.: Polyhedra scanning revisited. In: Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation, ser. PLDI 2012, pp. 499–508, June 2012
13. Adamski, D., Jablonski, G., Perek, P., Napieralski, A.: Polyhedral source-to-source compiler. In: 2016 MIXDES - 23rd International Conference Mixed Design of Integrated Circuits and Systems, pp. 458–463, June 2016

14. Grosser, T., Armin, G., Lengauer, C.: Pollyâperforming polyhedral optimizations on a low-level intermediate representation. Parallel Process. Lett. **22**(04), 1250010 (2012)
15. Baskaran, M.M., Ramanujam, J., Sadayappan, P.: Automatic C-to-CUDA code generation for affine programs. In: Proceedings of the International Conference on Compiler Construction, March 2010
16. Leung, A.: A mapping path for multi-GPGPU accelerated computers from a portable high level programming abstraction. In: Workshop on General-Purpose Processing using GPUs, September 2010
17. http://en.cppreference.com/w/cpp/container/vector
18. Sujeeth, A.K., et al.: Delite: a compiler architecture for performance-oriented embedded domain-specific languages. ACM Trans. Embed. Comput. Syst. **13**(4s), 134:1–134:25 (2014). https://doi.org/10.1145/2584665