Offload acceleration of scientific calculations within .NET assemblies*

Lebedev A.[1], Khachumov V.[2]

[1]*Rybinsk State Aviation Technical University, Rybinsk, Russia*
[2]*Institute for Systems Analysis of Russian Academy of Sciences, Moscow, Russia*

tementy@gmail.com, vmh48@mail.ru

Abstract.

A solution to improve portability of automatically parallelized code and extend applicability of polyhedron model is reached via on-the-fly transformations within JIT mechanisms of virtual execution system. An architecture of parallelizing optimizer module for ILDJIT is discussed. Target parallel architectures are many-integrated core (Intel MIC) and common x86 multicore processors. The LU-decomposition example written in C# illustrates significant speedup over traditional JIT-execution after being parallelized by execution system in runtime.

Keywords

Automatic parallelization, Polytope model, Accelerators, Many integrated core architecture, Just-in-time compilation, Managed code.

1 Introduction

Large number of processing elements per one chip is a common feature of modern microarchitectures. Many of them are represented by highly-parallel processors: graphics processing units (GPUs), many integrated core (Intel MIC) and universal multicore processors (CPUs). Writing an efficient code to expose compute capabilities of parallel processor often leads to a significant challenges. A very attractive approach addressing this issue is automatic parallelization (ideally, without any programmers effort).

Many compute-intensive applications often spend most of their execution time in nested loops. This is particularly common in scientific and engineering applications. The polyhedral model provides a powerful abstraction to reason about transformations on such loop nests consigned for locality and parallelization optimizations.

Polyhedron model methods are used commonly in static compilation scheme [1, 2, 3, 4, 5]: the code written in high-level language like C or FORTRAN is optimized once (for data locality, parallel execution) and runs on target architecture. Of course, native code compiled for target parallel architecture with aggressive optimization performs well. But, in general case, such a solution has drawbacks. Firstly, the code produced by native compiler may have portability issues (for example, Intel MIC and AMD64 are similar but incompatible instruction set architectures). In the second place stands a problem of applicability of polyhedron model to certain program: a program being analyzed must belong to linear class. But may occur a situation when not all model parameters can be derived in compile-time, i.e. program being analyzed don't belong to linear class (for example, indirect array addressing violates polyhedron model applicability).

We propose a solution to address both problems. An idea of virtual execution environment exposed in many current platforms similar to Java or .NET was initially developed to address code portability problem. Since modern JIT systems perform not only translation to machine-specific byte-code, but various optimizations as well, the parallelization optimizer seems to be a natural extension of common optimization set. The main advantage of shifting parallelization to runtime is extended possibilities of polyhedron model application - those of model parameters which values are unknown at compile-time can be determined in runtime leading to (possibly) linear program.

* Work was accomplished with hardware supplied by Intel Corporation to Rybinsk State Aviation Technical University.

2 An auto-parallelizing runtime

Currently, we consider two similar parallel architectures: well-known x86 multicore CPUs and recently appeared Intel MIC (Many Integrated Core). Firstly, we describe a mathematical background of our parallelizer - polyhedral framework. Then we show similarities of Intel MIC architecture to common CPUs. Later, the code optimization pipeline (slightly different for two architectures) is depicted. Lastly, an example is considered.

2.1 Overview of the polyhedral framework

The polytope model [6] is a computational model for sequential or parallel programs: a program is modelled by a polytope, a finite convex set of some dimensionality with flat surfaces. The data dependencies between the points (computations) in the polytope must be regular and local. Dependences are determined precisely through dataflow analysis, but we consider all dependences including anti (write-after-read), output (write-after-write) and input (read-after-read) dependences. These properties required of the model (convexity, flat surfaces, regularity of the dependencies) restrict the set of programs that can be modeled (linear programs). A set of, say, r perfectly nested loops with a constant-time loop statement, with bounds that are linear expressions in the indexes of the enclosing loops and in the problem size, and with certain additional restrictions on the use of the loop indexes can be represented by a polytope embedded in Z^r each loop defines the extent of the polytope in one dimension.

The Data Dependence Graph (DDG) is a directed multi-graph with each vertex representing a statement, and an edge, $e \in E$, from node S_i to S_j representing a polyhedral dependence from a dynamic instance of S_i to one of S_j: it is characterized by a polyhedron, P_e, called the dependence polyhedron that captures the exact dependence information corresponding to e. The dependence polyhedron is in the sum of the dimensionalities of the source and target statement's polyhedra (with dimensions for program parameters as well). Let s represent the source iteration and t be the target iteration pertaining to a dependence edge e. It is possible to express the source iteration as an affine function of the target iteration, i.e., to find the last conflicting access. This affine function is also known as the h-transformation, and will be represented by h_e for a dependence edge e. Hence, $s = h_e(t)$. The equalities corresponding to the h-transformation are a part of the dependence polyhedron and can be used to reduce its dimensionality. Let $S_1, S_2, ..., S_n$ be the statements of the program. A one-dimensional affine transform for statement S_k is defined by:

$$\phi_{S_k}(i) = [c_1, ..., c_{m_{S_k}}](i) + c_0 \tag{1}$$

ϕ_{S_k} can also be called an affine hyperplane, or a scattering function when dealing with the code generator. A multi-dimensional affine transformation for a statement is represented by a matrix with each row being an affine hyperplane.

When modelling sequential execution, the dimensions are scanned, one at a time, to enumerate the points in the polytope. For a parallel execution, the polytope that represents the source program – we call it the source polytope – is segmented into time slices, sets of points that can be executed concurrently. The parallelization methods based on the polytope model address the problem of this segmentation. It can be formulated as an affine mapping that transforms the source polytope into a target polytope that contains the same points, but in a new coordinate system in which some dimensions are strictly temporal, i.e., scanning along them enumerates time, and the others are strictly spatial, i.e., scanning along them enumerates space.

Consider the following affine form:

$$\delta_e(s,t) = \phi_{S_j}(t) - \phi_{S_i}(s), \langle s,t \rangle \in P_e \tag{2}$$

The affine form $\delta_e(s,t)$ is very significant. This function is the number of hyperplanes the dependence e traverses along the hyperplane normal ϕ. If ϕ is used as a space loop to generate tiles for parallelization, this function is a factor in the communication volume. On the other hand, if ϕ is used as a sequential loop, it gives us a measure of the reuse distance. An upper bound on this function would mean that the number of hyperplanes that would be communicated as a result of the dependence at the tile boundaries would not exceed the bound, the same for cache misses at L1/L2 tile edges, or L1 cache loads for a register tile. Of particular interest is, if this function can be reduced to a constant amount or zero (free of a parametric component) by choosing a suitable direction for ϕ: if this is possible, then that particular dependence leads to constant boundary communication or no communication (respectively) for this hyperplane. We rely on approach of minimizing data reuse distance $\delta_e(s,t)$ for all dependencies to obtain good schedule [4].

2.2 Many Integrated Core Architecture

Intel Many Integrated Core architecture [7] combines many Intel CPU cores onto a single chip. Intel MIC architecture is targeted for highly parallel, High Performance Computing workloads in a variety of fields. A key attribute of the microarchitecture is that it is built to provide a general-purpose programming environment similar to the Intel Xeon processor programming environment. The Intel Xeon Phi coprocessor is connected to an Intel Xeon processor, also known as the host, through a PCI Express bus. Since the Intel Xeon Phi coprocessor runs a Linux operating system, a virtualized TCP/IP stack is implemented over the PCIe bus, allowing the user to access the coprocessor as a network node. Thus, any user can connect to the coprocessor through a secure shell and directly run individual jobs or submit batchjobs to it.

The Intel Xeon Phi coprocessor is primarily composed of processing cores, caches, memory controllers, PCIe client logic, and a very high bandwidth, bidirectional ring interconnect (figure 1). Each core comes complete with a private L2 cache that is kept fully coherent by a global-distributed tag directory. The memory controllers and the PCIe client logic provide a direct interface to the GDDR5 memory on the coprocessor and the PCIe bus, respectively. All these components are connected together by the ring interconnect.

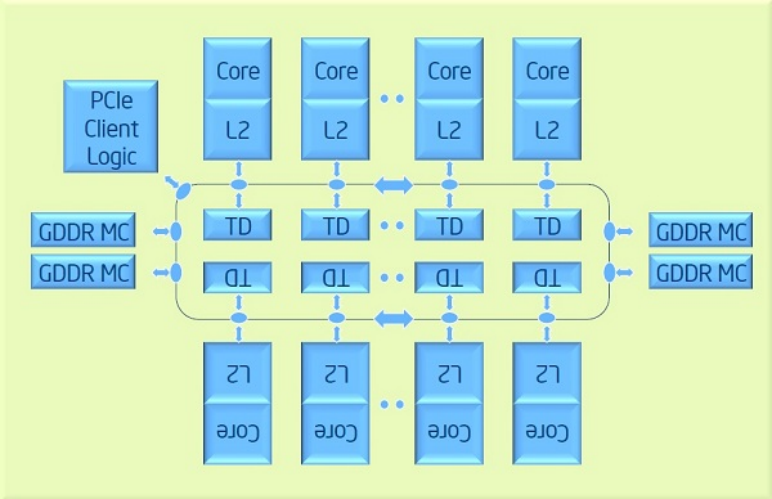

Figure 1. Intel Xeon Phi microarchitecture: cores, interconnect, caches, memory controllers.

An important component of the Intel Xeon Phi core is vector processing unit (VPU). The VPU features a novel 512-bit SIMD instruction set, officially known as Intel Initial Many Core Instructions. Thus, the VPU can execute 16 single-precision (SP) or 8 double-precision (DP) operations per cycle. The VPU also supports Fused Multiply-Add (FMA) instructions and hence can execute 32 SP or 16 DP floating point operations per cycle. It also provides support for integers.

Widely adopted programming techniques such as POSIX threads, OpenMP, MPI are applicable to Xeon Phi as well, because it is actually an x86 SMP-on-a-chip running Linux. Such a similarity to common x86 systems allows us to conclude that well-known parallelization methods developed for multi-core processors would be fruitful being applied to many-integrated core domain.

2.3 Parallelization as an optimization phase in JIT-oriented runtime

We use Intermediate Language Distributed Just In Time (ILDJIT) [8], a compilation framework for .NET designed to be both easily extensible and easily configurable, and develop parallelization extensions to it. ILDJIT provides two different dynamic compilation schemes: the just-in-time (JIT) and the dynamic-look-ahead (DLA) one. The second one, the DLA compilation, is a natural evolution of the JIT compilation specifically designed for the multicore era. These schemes interchange the compilation and the execution of the program leading to an interleaving of the corresponding phases shown in figure 2.

An overall scheme of code optimization pipeline is shown in figure 3. We have integrated state-of-art optimization and analysis approaches based on polyhedron model to the optimizer. A code to be parallelized is stored initially in ILDJIT IR (a representation ILDJIT basically works with after instruction selection phase finishes) and needs to be parsed with

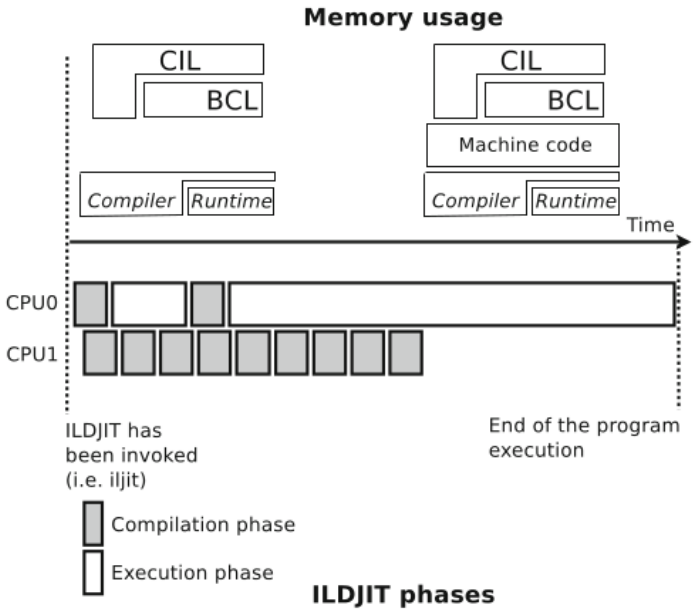

Figure 2. Dynamic look-ahead compilation scheme implemented inside ILDJIT.

polyhedron extractor: for loops with induction variables, statements with (necessary) affine indexes are recognized and then transformed into polyhedral definition of statement domains called Polyhedral IR (P. IR). Dependence polyhedrons are extracted with candl [9]. Optimal schedule in case of reuse distance minimization is obtained with pluto framework [4]. Code generation is performed with slightly modified version of CLooG [10]: for general x86 case the ILDJIT IR is generated, but for Intel MIC standard C code is the option. Parallel loops are marked with OpenMP pragmas in C code (with the help of pluto component) and are instrumented with POSIX thread calls in ILDJIT IR to expose OpenMP-like iterations scheduling. Since ICC compiler is the only way to get machine code for Intel MIC for now, the compilation phase for it finishes with native code library obtained from code instrumented with special ICC offload pragmas. Shared objects (.so for Intel MIC) and ILDJIT IR subprograms with their native representations obtained in runtime as a result of JIT compilation are stored in on-disk database called compute cache along with corresponding model parameters value set. This is done at the first time call to avoid unnecessary recompilation for already processed subprogram and its parameters. When a parallel subprogram is called, then a shared-library call is performed for Intel MIC, or in-memory generated code is called for x86.

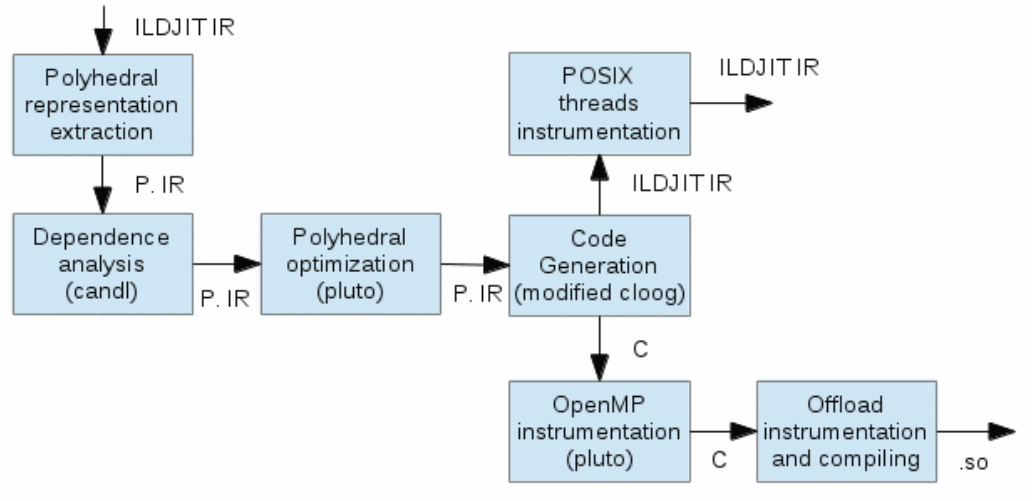

Figure 3. Phases of parallelization added to JIT mechanism of ILDJIT.

2.4 The example

We consider simple LU-decomposition kernel and compare several variants of it: naive parallelization with OpenMP, optimized for reuse distance, optimized for reuse distance and tiled with certain shape for optimal cache usage (parallel C code marked with PLUTO in comments was obtained from Intel MIC compilation pipeline):

```
// C# version:
for (int k = 0; k < N; k++) {
    for (int j = k + 1; j < N; j++)
        A[j * N + k] /= A[k * N + k];
    for (int i = k + 1; i < N; i++)
        for (int j = k + 1; j < N; j++)
            A[i * N + j] -= A[i * N + k] *
                A[k * N + j];
}

//Naive OpenMP C version:
int i, j, k;
for (k = 0; k < N; k++) {
#pragma omp parallel for
    for (j = k + 1; j < N; j++)
        A[j * N + k] /= A[k * N + k];
#pragma omp parallel for private(j)
    for (i = k + 1; i < N; i++)
        for (j = k + 1; j < N; j++)
            A[i * N + j] -= A[i * N + k] *
                A[k * N + j];
}
```

```
// PLUTO, C version without tiling:
int t1, t2, t3;
int lb, ub, lbp, ubp, lb2, ub2;
register int lbv, ubv;
/* Start of CLooG code */
if (N >= 2) {
  for (t1=1;t1<=2*N-3;t1++) {
    lbp=ceild(t1+1,2);
    ubp=min(t1,N-1);
#pragma omp parallel for private(lbv,ubv,t2,t3)
    for (t2=lbp;t2<=ubp;t2++) {
      a[t2][t1-t2]=a[t2][t1-t2]/a[t1-t2][t1-t2];;
      for (t3=t1-t2+1;t3<=N-1;t3++) {
        a[t2][t3]=a[t2][t3]-
            a[t2][t1-t2]*a[t1-t2][t3];;
      }
    }
  }
}
/* End of CLooG code */
```

```
// PLUTO, C version with 64x64x64 tile size:
int t1, t2, t3, t4, t5, t6;
int lb, ub, lbp, ubp, lb2, ub2;
register int lbv, ubv;
/* Start of CLooG code */
if (N >= 2) {
  for (t1=0;t1<=floord(N-2,32);t1++) {
    lbp=ceild(t1,2);
    ubp=min(floord(N-1,64),t1);
#pragma omp parallel for private(lbv,ubv,t2,t3,t4,t5,t6)
    for (t2=lbp;t2<=ubp;t2++)
      for (t3=t1-t2;t3<=floord(N-1,64);t3++) {
        if (t1 == t2+t3)
          for (t4=64*t1-64*t2;t4<=min(N-2,64*t1-64*t2+62);t4++)
            for (t5=max(64*t2,t4+1);t5<=min(N-1,64*t2+63);t5++) {
              a[t5][t4]=a[t5][t4]/a[t4][t4];;
#pragma ivdep
              for (t6=t4+1;t6<=min(N-1,64*t1-64*t2+63);t6++)
                a[t5][t6]=a[t5][t6]-a[t5][t4]*a[t4][t6];;
            }
        if ((t1 == t2+t3) && (t1 <= 2*t2-1))
          for (t5=64*t2;t5<=min(N-1,64*t2+63);t5++)
            a[t5][64*t1-64*t2+63]=a[t5][64*t1-64*t2+63]/a[64*t1-64*t2+63][64*t1-64*t2+63];;
        if (t1 <= t2+t3-1)
          for (t4=64*t1-64*t2;t4<=min(64*t2+62,64*t1-64*t2+63);t4++)
            for (t5=max(64*t2,t4+1);t5<=min(N-1,64*t2+63);t5++)
#pragma ivdep
              for (t6=64*t3;t6<=min(N-1,64*t3+63);t6++)
                a[t5][t6]=a[t5][t6]-a[t5][t4]*a[t4][t6];;
      }
  }
}
/* End of CLooG code */
```

Table 1 shows time (given in seconds) of LU-kernel execution for 2000x2000 float matrix. As we can see, direct naive parallelization outperforms solutions obtained by reuse distance minimization.

Table 1. Results of LU-kernel benchmarks.

	1x1x1	2x2x2	4x4x4	8x8x8	16x16x16	32x32x32	64x64x64	Naive
PHI 3100 228 threads	1.01	3.64	2.34	1.93	1.48	1.47	1.97	0.65
PHI 3100 1 thread	45.44	205.52	85.03	39.00	18.16	9.83	7.66	6.61
E2690 8 threads	2.22	3.59	2.48	2.31	2.32	2.59	2.95	1.66
E2690 1 thread	13.16	21.53	14.62	13.23	12.50	12.77	12.45	11.58

Table 2 shows acceleration factor meaning speedup (in times) achieved by using all of chip cores comparably to only one for each algorithm. While any considered algorithm preserves good scalability, the naive one performs best. This means that modern microarchitectures like Sandy Bridge and Many Integrated Core expose more complex cache policies which need to be considered in optimization. But any of solutions found by parallelizer is worth parallelization overhead: for Xeon Phi 3100 speedup is between 1.8 and 6.6 times; for Xeon E2690 speedup is between 3.2 and 5.2 times.

Table 2. Results of LU-kernel benchmarks.

	1x1x1	2x2x2	4x4x4	8x8x8	16x16x16	32x32x32	64x64x64	Naive
PHI acceleration	45.16	56.40	36.27	20.21	12.24	6.70	3.90	10.20
E2690 acceleration	5.93	5.99	5.89	5.72	5.39	4.93	4.22	7.00

3 Conclusion

A solution provided here allows to write sequential code once using high-level language (we consider C# for now) and then run it in parallel on different architectures without full manual recompilation. Significant speedup is achieved with polyhedral parallelization, but underlying framework needs to be revised in order to expose compute capabilities of new hardware.

References

[1] Irigoin, Francois, Pierre Jouvelot, and Rémi Triolet. Semantical interprocedural parallelization: An overview of the PIPS project. *Proceedings of the 5th international conference on Supercomputing*. ACM, 1991.

[2] Lee, Sang-Ik, Troy A. Johnson, and Rudolf Eigenmann. Cetus–an extensible compiler infrastructure for source-to-source transformation. *Languages and Compilers for Parallel Computing*. Springer Berlin Heidelberg, 2004. 539-553.

[3] Griebl, Martin, and Christian Lengauer. The loop parallelizer LooPo. *Proc. Sixth Workshop on Compilers for Parallel Computers*. Vol. 21. Konferenzen des Forschungszentrums Jülich, 1996.

[4] Bondhugula, Uday, et al. Automatic transformations for communication-minimized parallelization and locality optimization in the polyhedral model. *Compiler Construction*. Springer Berlin Heidelberg, 2008.

[5] Grosser, Tobias, et al. Polly-Polyhedral optimization in LLVM. *Proceedings of the First International Workshop on Polyhedral Compilation Techniques (IMPACT)*. Vol. 2011. 2011.

[6] Christian Lengauer. Loop Parallelization in the Polytope Model. In Eike Best, editor, CONCUR'93, number 715 in Lecture Notes in Computer Science, pages 398–416. Springer-Verlag, 1993.

[7] http://software.intel.com/en-us/articles/intel-xeon-phi-coprocessor-codename-knights-corner

[8] Campanoni, Simone. Guide to ILDJIT. Springer, 2011.

[9] http://www.cs.ucla.edu/pouchet/software/pocc/

[10] Bastoul, Cedric. Code generation in the polyhedral model is easier than you think. *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*. IEEE Computer Society, 2004.