Dynamic Parallelization of Computational Code as a Phase of Just-in-Time Compilation

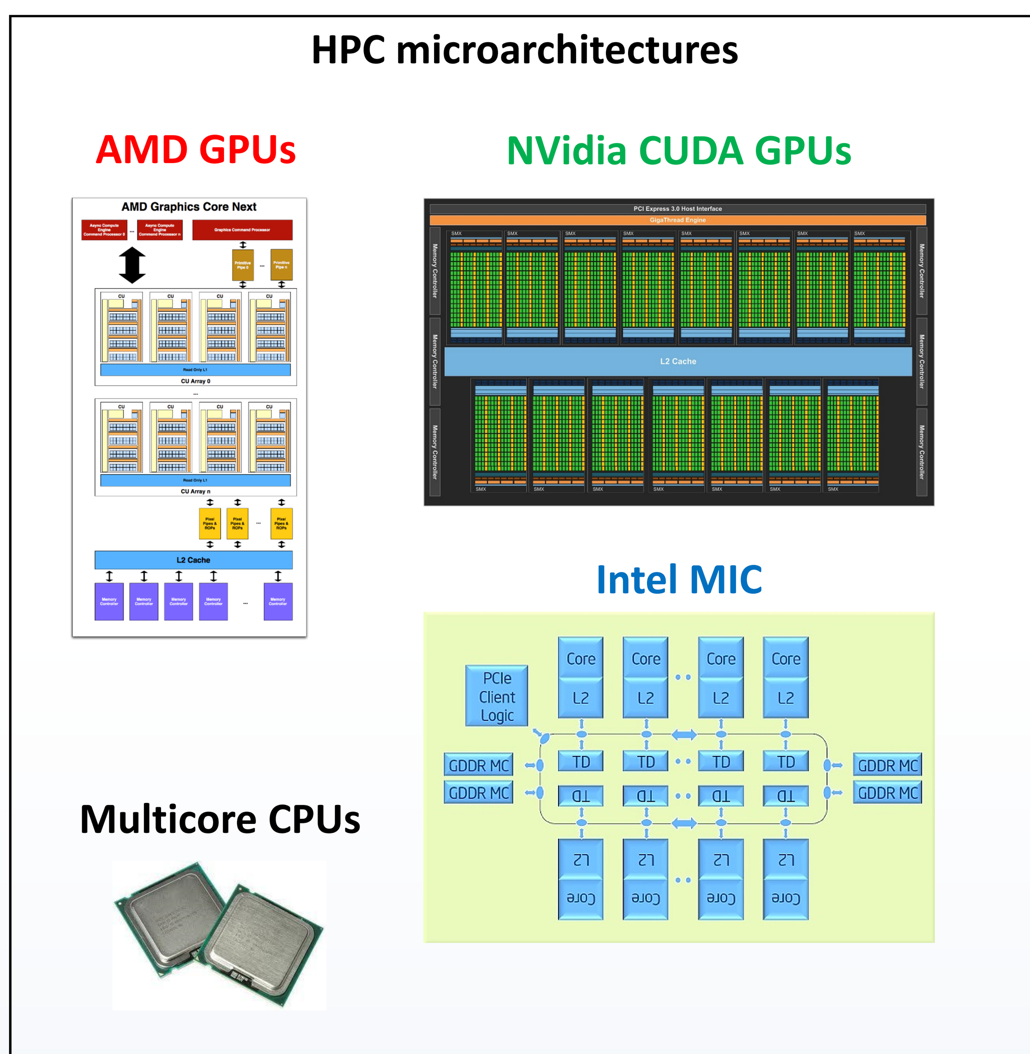
A. Lebedev
Rybinsk State Aviation Technical University, Rybinsk, Russia
tementy@gmail.com

Abstract

A solution to improve portability of automatically parallelized code, extend applicability of polyhedron model, organize dynamic load balancing between host system and accelerator is reached via on-the-fly transformations within JIT mechanisms of virtual execution system. An architecture of parallelizing optimizer module for ILDJIT is discussed. Transformed for heterogeneous execution LU-decomposition kernel illustrates significant speedup due to offload acceleration.

Motivation

Writing an efficient code to expose compute capabilities of parallel processor often leads to significant challenges. The task becomes more complex when targeting heterogeneous computing systems. The project aims to simplify programming of modern parallel architectures: multicore CPUs and various accelerators, such as AMD/NVidia GPUs and Intel Xeon Phi.



A very attractive approach addressing the issue is automatic code parallelization. There is a variety of modern and mature parallelizing compilers – commercial (PGI, CAPS, Intel) and free (LOOPO, PLUTO, PIPS, CETUS). We state following facts for them:

- Polyhedron model is a common mathematical background for open-source solutions.
- All the compilers perform static compilation (code is compiled once for a particular parallel microarchitecture).

The second position reveals perceptive drawbacks of commonly used approach:

- Binary image is not portable w. r. t. parallel architectures.
- Too difficult organization of load balancing between host and accelerator.
- There could be difficulties in polyhedron model applying: not all model parameters may be defined statically or derived in compile-time.

Goals

We rely on strong mathematical background of polyhedron model and work for improving of common practices of using parallelizing compilers to:

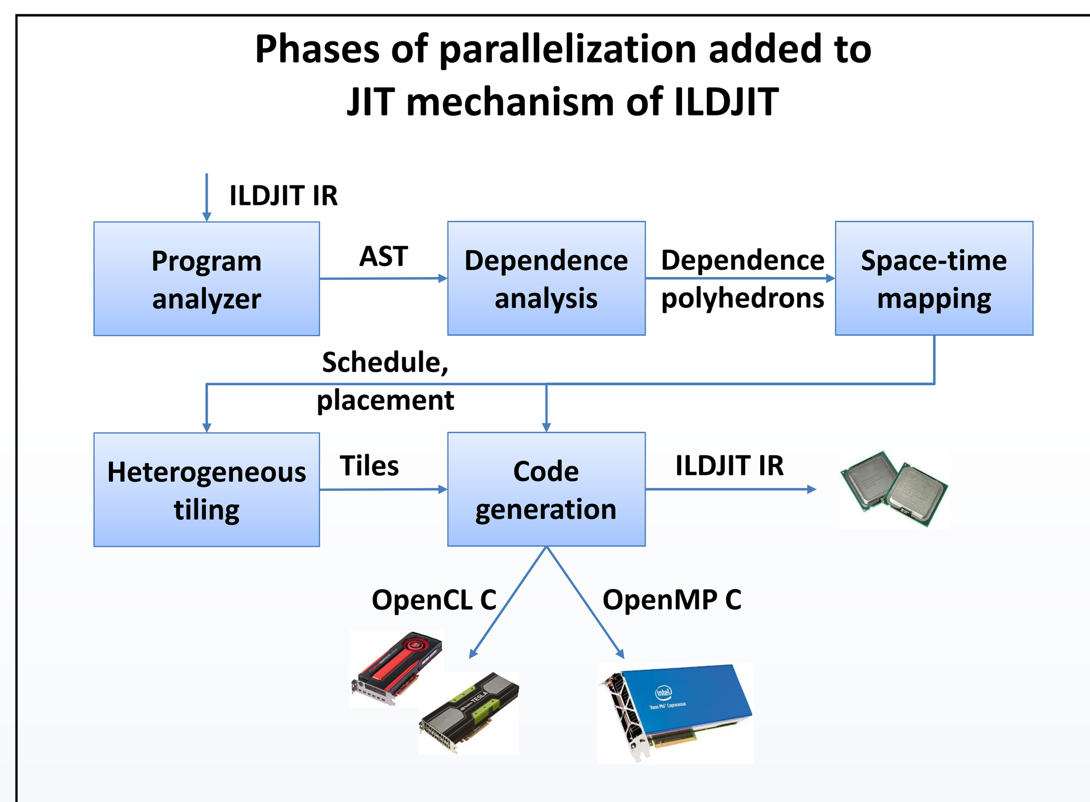
- reach the portability of automatically parallelized code w. r. t. parallel architectures - the code written once must be runnable on several (supported) parallel architectures without full manual recompilation.
- shift the polyhedron model use to the extreme to deal with statically undefined model parameters.

An auto-parallelizing runtime

We propose a solution based on idea of virtual execution environment to address stated problems.

- All the parameters (technically - variables) become defined in runtime - the polyhedron model could be applied to wider set of programs, not only strict linear ones.
- Code portability is now the problem of execution environment, not the programmer.
- Platform-specific optimizations and heuristics could be applied within JIT mechanism in order to get best performance of parallelized code.

We chose ILDJIT – an open-source .NET virtual execution system and develop parallelizing extension for it.
We have integrated state-of-art optimization and analysis approaches based on polyhedron model to the optimizer.
A code to be parallelized needs to be parsed with polyhedron extractor firstly: for loops with induction variables, statements with (necessary) affine indexes are recognized and then transformed into polyhedral definition of statement domains. Space-time mapping leading to maximal parallelism is then computed.
Resulting parallel programs are saved in on-disk storage called compute cache at the first time call to avoid unnecessary recompilation for already processed subprogram and its parameters in future.



Dynamic load balancing

A heuristic is to choose space-time mapping uniformly exposing fine-grained parallelism w.r.t. most of time steps (if possible), then determine representative benchmark tile size and estimate time parameters in order to compute α value for every time step. The code generation phase should be followed by instrumentation: runtime estimations and dynamic workload distribution should be injected into target program.

- A portion of virtual processors (say, α from N) on every time step should be mapped to accelerator, if reasonable. We also should leave one CPU core for accelerator management. Rest $N-\alpha$ virtual processors should be mapped to free CPU cores.
- Let $T_{ACC} = dep_{cost}^{input} + \alpha * vp_{cost}^{ACC}$ be accelerator time, where dep_{cost}^{input} stands for input dependencies communication overhead and vp_{cost}^{ACC} is average processing cost of computations corresponding to one virtual processor, including communication.
- Let $T_{CPU} = \frac{N-\alpha-probed}{NP-1} * vp_{cost}^{CPU}$ be CPU time, where NP is the number of CPU cores, vp_{cost}^{CPU} is above, $probed$ is the number of slices processed while estimating time parameters.
- All the parameters we suggest to estimate empirically. Total execution cost is $\max(T_{ACC}, T_{CPU})$. It means that we look for the point α for which T_{ACC}=T_{CPU} holds. Usage of the accelerator is reasonable if and only if α>1.

$$\alpha = \frac{(N - probed) * vp_{cost}^{CPU} - (NP-1) * dep_{cost}^{input}}{vp_{cost}^{ACC} * (NP-1) + vp_{cost}^{CPU}}$$

LU-decomposition example

$A = LU$,

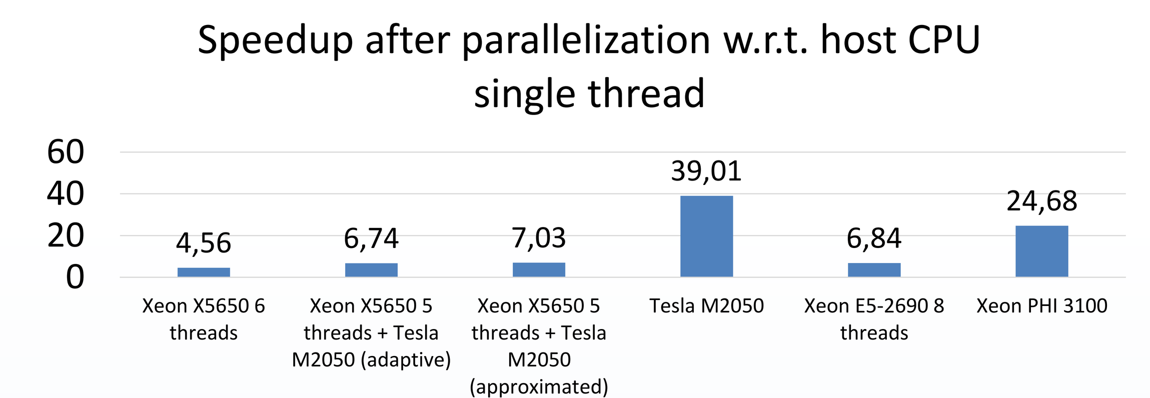
$$\begin{bmatrix} a_{1,1} & a_{1,2} & a_{1,3} \\ a_{2,1} & a_{2,2} & a_{2,3} \\ a_{3,1} & a_{3,2} & a_{3,3} \end{bmatrix} = \begin{bmatrix} l_{1,1} & 0 & 0 \\ l_{2,1} & l_{2,2} & 0 \\ l_{3,1} & l_{3,2} & l_{3,3} \end{bmatrix} \begin{bmatrix} u_{1,1} & u_{1,2} & u_{1,3} \\ 0 & u_{2,2} & u_{2,3} \\ 0 & 0 & u_{3,3} \end{bmatrix}.$$

C# version:

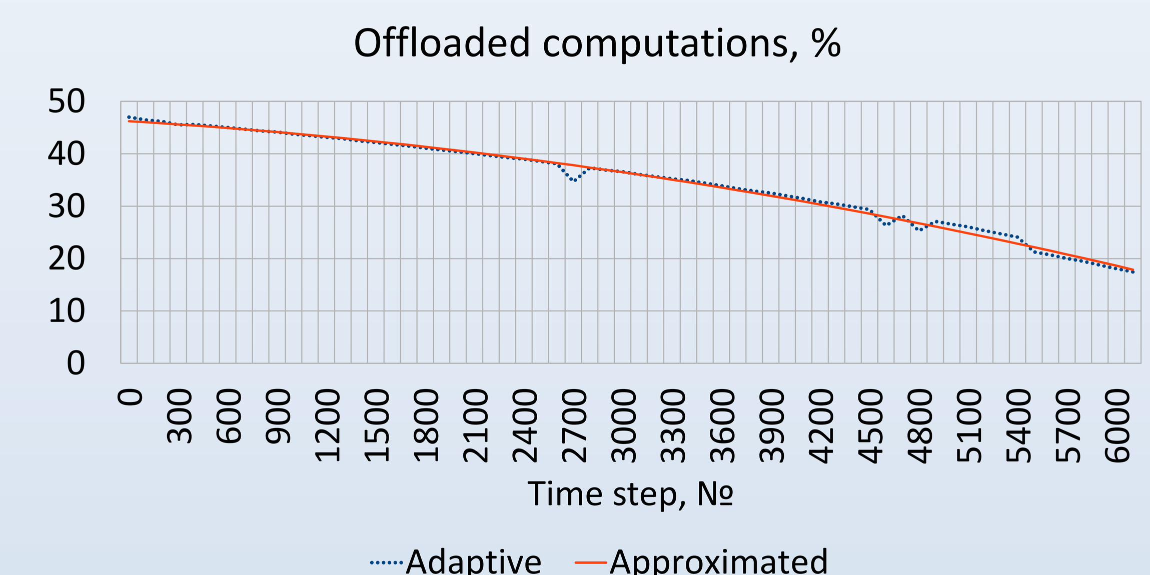
```
for (int k=0; k<N; k++)
{
  for (int j=k+1; j<N; j++)
    A[j][k] /= A[k][k];
  for (int i=k+1; i<N; i++)
    for (int j=k+1; j<N; j++)
      A[i][j] -= A[i][k]*A[k][j];
}
```

Underlined loops are parallel – their iterations could be mapped to different virtual and physical processors.
k-loop is considered as time-loop.
We made benchmarks for square matrix of order 2^{13} of doubles on two different heterogeneous computing systems:

- Intel Xeon X5650 + NVidia Tesla M2050
  - Parallelization for CPU cores
  - Dynamic load balancing between CPU cores and GPU
  - Pure GPU offload
- Intel Xeon-E5 2690 + Intel Xeon Phi 3100
  - Parallelization for CPU cores
  - Pure Xeon Phi offload



For dynamic load balancing case we collected values of alpha during run time (adaptive variant) and tried to approximate the regression with quadratic polynomial (approximated variant):
$-4.5 * 10^{-7} * k^2 - 0.00189496 * k + 46.22040315$.
Approximated variant performs faster because of eliminated estimations. Unfortunately, we had no success with Xeon PHI system – latencies of data transition were too large when compared to NVidia platform.



Results

- The solution 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 and adaptive load balancing between host and accelerator on every time step.

One can benefit from offloading of computations onto modern accelerators; but should be careful with scenarios with intensive data transition over PCI-E bus. While NVidia platform allows dynamic load balancing, this is not an option because pure offload could perform better. Xeon PHI platform works well with pure offload and big data transitions hiding huge latencies.

Acknowledgements

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