# Performance Portable Parallel Programming

## Compile-Time Defined Parallelization and Storage Order for Accelerators and CPUs

**Michel Müller**
MSc ETH Zurich
**michel@typhooncomputing.com**
Aoki Laboratory, GSIC, Tokyo Institute of Technology

## Abstract

Performance portability between CPU and accelerators is a major challenge for coarse grain parallelized codes. Hybrid Fortran offers a new approach in porting for accelerators that requires minimal code changes and allows to keep the performance of CPU optimized loop structures and storage orders. This is achieved through a compile-time code transformation where the CPU and accelerator cases are treated separately. Results show minimal performance losses compared to the fastest non-portable solution on both CPU and GPU. Using this approach, five applications have been ported to accelerators, showing minimal or no slowdown on CPU while enabling high speedups on GPU.

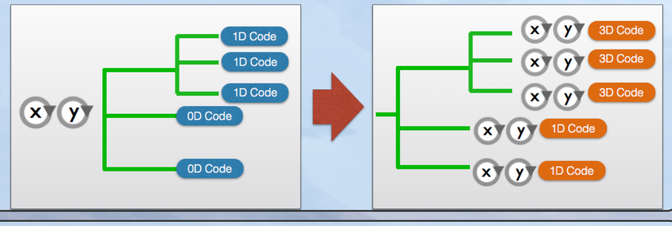
## 1. Motivation

When porting real world HPC applications for accelerators, performance portability is often one of the main goals - it is imperative that code can be executed on different architectures with at least reasonable performance. Achieving this for accelerators is a major challenge since their architecture is so different from CPUs.

Often the biggest change is going from coarse grained parallelism (order of 10-100 threads per processor) to fine grained parallelism (order of 10'000 - 100'000 threads per processor). This is particularly challenging for code that is parallelized at a point in the program that is far removed from the actual computations. The most prominent example are physical cores for weather and climate models.

The usual approach is to privatize the code in the parallel domains, such that it can be split up into multiple smaller kernels. This leads to problems:

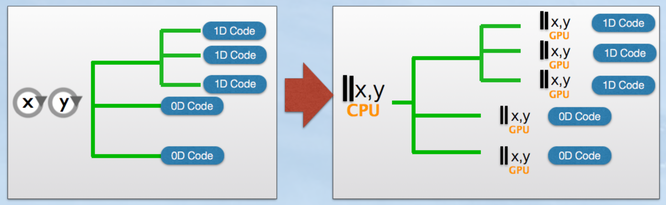
1. When executing this code on CPUs, it usually leads to substantial performance losses. It is therefore not performance portable.
2. It leads to a complete rewrite of the computational code, with lots of mechanical work for simply inserting additional domains in declarations and accessors. This is bug prone and leads to less readable code.




## 2. Proposal

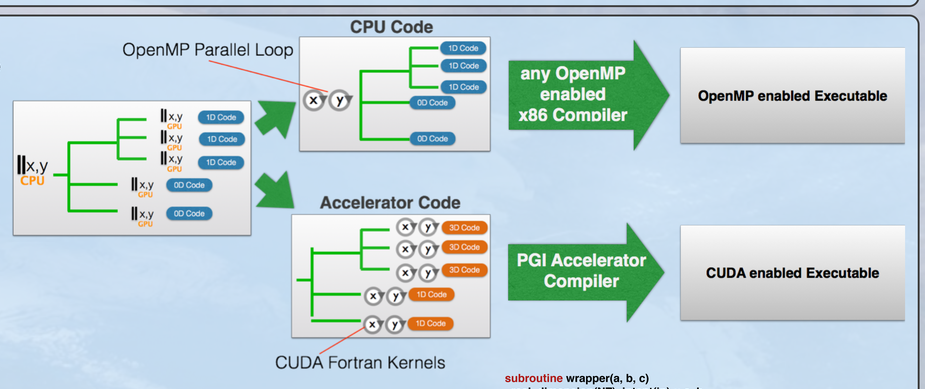
In order to (1) ensure performance portability and (2) minimize code portation, we propose the following solution:

1. Allow both coarse grained and fine grained parallelization in the same codebase through directives. This enables optimal parallelization for both CPU and accelerator architectures.
2. Automate the privatization of symbols where needed, such that the original code can be kept with a low number of dimensions.




## 3. Method

**Hybrid Fortran**[1] is an **Open Source preprocessor framework** and a **Fortran language extension** developed for the task of allowing such hybridized parallelizations as described in (2) and transforming such unified codes into standard x86 Fortran and Accelerator enabled Fortran. So far, OpenMP and CUDA Fortran parallelizations are implemented. Hybrid Fortran currently supports **any data parallel code that can be implemented on shared memory systems.** Storage order is abstracted and can be defined in a central location without any changes to array accessors and declarations.




Specify symbols a,b,c to be privatized in X,Y when needed

CPU Parallelized outside procedure in X,Y

Fortran Declarations - No changes needed

GPU Parallelized inside procedure in X,Y

**1D Computational Code (in Z) - No changes needed**

```
subroutine wrapper(a, b, c)
  real, dimension(NZ), intent(in) :: a, b
  real, dimension(NZ), intent(out) :: c
  @domainDependant{domName(x,y), domSize(NX,NY), attribute(autoDom)}
  a, b, c
  @end domainDependant
  @parallelRegion{appliesTo(CPU), domName(x,y), domSize(NX, NY)}
  call add(a, b, c)
  @end parallelRegion
end subroutine
subroutine add(a, b, c)
  real, dimension(NZ), intent(in) :: a, b
  real, dimension(NZ), intent(out) :: c
  integer :: z
  @domainDependant{domName(x,y), domSize(NX,NY), attribute(autoDom)}
  a, b, c
  @end domainDependant
  @parallelRegion{appliesTo(GPU), domName(x,y), domSize(NX, NY)}
  do z=1,NZ
     c(z) = a(z) + b(z)
  end do
  @end parallelRegion
end subroutine
```

## 4. Performance Results

| | Performance Characteristic | Speedup HF on 6 Core vs. 1 Core [A] | Speedup HF on GPU vs 6 Core [A] | Speedup HF on GPU vs 1 Core [A] |
|---|---|---|---|---|
| **1.** ASUCA Physical Weather Prediction Core (121 Kernels) [2] | Mixed, Coarse Grain Parallelism | 4.47x | 3.63x | 16.22x |
| **2.** 3D Diffusion (Source on Github) [1] | Memory Bandwidth Bound, Fine Grained Parallelism | 1.06x | 10.94x | 11.66x |
| **3.** Particle Push (Source on Github) [1] | Computationally Bound, Sine/ Cosine operations, Fine Grained Parallelism | 9.08x | 21.72x | 152.79x |
| 4. Poisson on FEM Solver with Jacobi Approximation (Source on Github) [1] | Memory Bandwidth Bound, Fine Grained Parallelism | 1.41x | 5.13x | 7.28x |
| 5. MIDACO Ant Colony Solver with MINLP Example (Source on Github) [1] [3] | Computationally Bound, Divisions, Coarse Grain Parallelism | 5.26x | 10.07x | 52.99x |

**[A]**
If available, comparing to reference C version, otherwise comparing to Hybrid Fortran CPU implementation.

Kepler K20x has been used as GPU if not stated otherwise, Westmere Xeon X5670 has been used as CPU (TSUBAME 2.5).

All results measured in double precision.

The CPU cores have been limited to one socket using thread affinity 'compact' with 12 logical threads.

For CPU, Intel compilers ifort / icc with '-fast' setting have been used.

For GPU, PGI compiler with '-fast' setting and CUDA compute capability 3.x has been used.

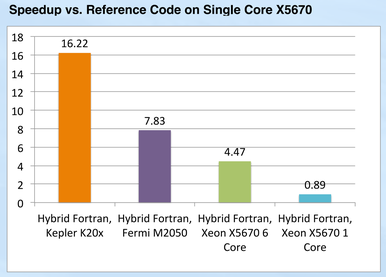
All GPU results include the memory copy time from host to device.

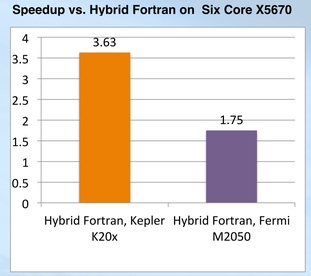
### 4.1 ASUCA Physical Weather Prediction Core Test

Radiation (SW + LW)
+ Planetary Boundary Layer
+ Surface Test

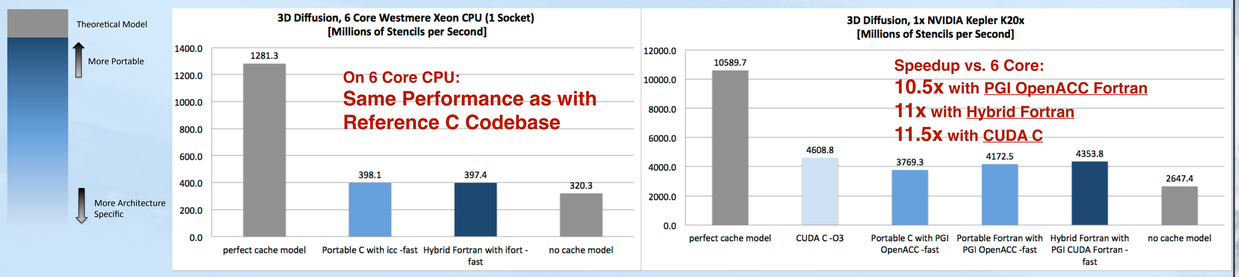
NI = 128, NJ = 128, NK = 70
100 Timesteps

**Speedup vs. 6 Core:**
**3.63x with Hybrid Fortran**

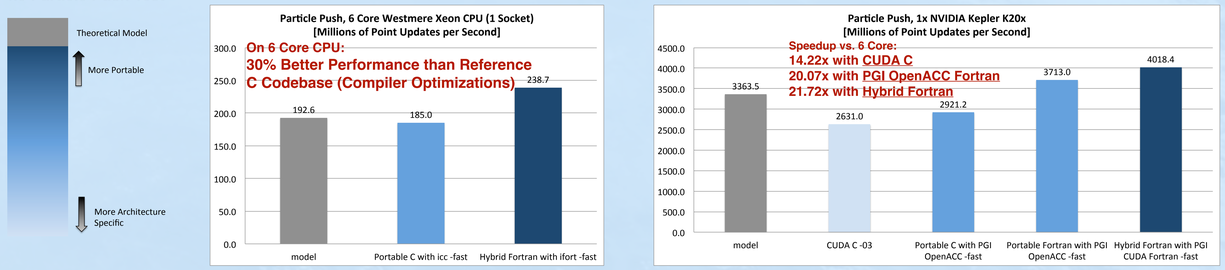



### 4.2 3D Diffusion Test




### 4.3 Particle Push Test

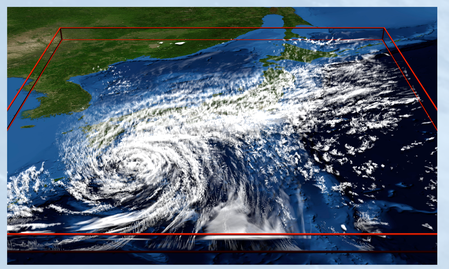



## 6. Conclusion and Future Work

The preprocessor framework "Hybrid Fortran" has been developed and shown to ..
1. .. be performance portable,
2. .. require minimum code changes for porting CPU code to accelerators,
3. .. be general purpose capable for various data parallel problems.

We will extend the work on ASUCA as well as other weather- and climate models. ASUCA on Hybrid Fortran will become a production ready weather model.

Future work includes
• Intel MIC support
• Support for Derived Types



ASUCA on 500m grid resolution [4]

**Supervised by**
Dr. Takashi Shimokawabe, Tokyo Institute of Technology
Prof. Dr. Aoki, Tokyo Institute of Technology
**Special Thanks to**
Dr. Martin Schlueter, MIDACO-Solver
Dr. Johan Hysing, Tokyo Institute of Technology

Tokyo Institute of Technology
GSIC Global Scientific Information and Computing Center
Typhoon Computing

Background: NASA

[1] M. Müller "Hybrid Fortran Github Repository", [Website, Updated 2014-7-24]
http://github.com/muellermichel/Hybrid-Fortran
[2] T. Hara et. al "Development of the Physics Library and its application to ASUCA", 2012
[3] M. Schlueter "MIDACO-Global Optimization Software for Mixed Integer Nonlinear Programming", 2009
[4] T. Shimokawabe, T. Aoki et. al "145 TFlops Performance on 3990 GPUs of TSUBAME 2.0 Supercomputer for an Operational Weather Prediction", 2011