CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
*Concurrency Computat.: Pract. Exper.* (2015)
Published online in Wiley Online Library (wileyonlinelibrary.com). DOI: 10.1002/cpe.3522

SPECIAL ISSUE PAPER

# An analysis of the feasibility and benefits of GPU/multicore acceleration of the Weather Research and Forecasting model

Wim Vanderbauwhede[1,*,†] and Tetsuya Takemi[2]

[1] *School of Computing Science, University of Glasgow, Glasgow, UK*
[2] *Disaster Prevention Research Institute, Kyoto University, Uji, Kyoto, Japan*

## SUMMARY

There is a growing need for ever more accurate climate and weather simulations to be delivered in shorter timescales, in particular, to guard against severe weather events such as hurricanes and heavy rainfall. Due to climate change, the severity and frequency of such events – and thus the economic impact – are set to rise dramatically. Hardware acceleration using graphics processing units (GPUs) or Field-Programmable Gate Arrays (FPGAs) could potentially result in much reduced run times or higher accuracy simulations. In this paper, we present the results of a study of the Weather Research and Forecasting (WRF) model undertaken in order to assess if GPU and multicore acceleration of this type of numerical weather prediction (NWP) code is both feasible and worthwhile. The focus of this paper is on acceleration of code running on a single compute node through offloading of parts of the code to an accelerator such as a GPU. The governing equations set of the WRF model is based on the compressible, non-hydrostatic atmospheric motion with multi-physics processes. We put this work into context by discussing its more general applicability to multi-physics fluid dynamics codes: in many fluid dynamics codes, the numerical schemes of the advection terms are based on finite differences between neighboring cells, similar to the WRF code. For fluid systems including multi-physics processes, there are many calls to these advection routines. This class of numerical codes will benefit from hardware acceleration. We studied the performance of the original code of the WRF model and proposed a simple model for comparing multicore CPU and GPU performance. Based on the results of extensive profiling of representative WRF runs, we focused on the acceleration of the scalar advection module. We discuss the implementation of this module as a data-parallel kernel in both OpenCL and OpenMP. We show that our data-parallel kernel version of the scalar advection module runs up to seven times faster on the GPU compared with the original code on the CPU. However, as the data transfer cost between GPU and CPU is very high (as shown by our analysis), there is only a small speed-up (two times) for the fully integrated code. We show that it would be possible to offset the data transfer cost through GPU acceleration of a larger portion of the dynamics code. In order to carry out this research, we also developed an extensible software system for integrating OpenCL code into large Fortran code bases such as WRF. This is one of the main contributions of our work. We discuss the system to show how it allows the replacement of the sections of the original codebase with their OpenCL counterparts with minimal changes – literally only a few lines – to the original code. Our final assessment is that, even with the current system architectures, accelerating WRF – and hence also other, similar types of multi-physics fluid dynamics codes – with a factor of up to five times is definitely an achievable goal. Accelerating multi-physics fluid dynamics codes including NWP codes is vital for its application to weather forecasting, environmental pollution warning, and emergency response to the dispersion of hazardous materials. Implementing hardware acceleration capability for fluid dynamics and NWP codes is a prerequisite for up-to-date and future computer architectures. Copyright © 2015 John Wiley & Sons, Ltd.

Received 20 December 2013; Revised 12 January 2015; Accepted 26 March 2015

KEY WORDS: general-purpose computation on graphics processing units (GPGPU); parallelization of simulation; large-scale scientific computing

*Correspondence to: Wim Vanderbauwhede, School of Computing Science, University of Glasgow, G12 8QQ Glasgow, UK.

†E-mail: wim.vanderbauwhede@glasgow.ac.uk

# 1. BACKGROUND

There is a growing need for ever more accurate climate and weather simulations to be delivered in shorter timescales, in particular, to guard against severe weather events such as hurricanes and heavy rainfall. Due to climate change, the severity and frequency of such events – and thus the economic impact – are set to rise dramatically [1, 2]. Hardware acceleration using graphics processing units (GPUs) or FPGAs could potentially result in much reduced run times or higher accuracy simulations. As climate change will result in more and more severe extreme weather events, faster, more accurate predictions of extreme weather events are needed [3, 4]. Understanding climate change itself requires growing amounts of computational power [5].

## *1.1. The weather research and forecasting model*

The Weather Research and Forecasting model[‡] (WRF) [6–8] is a state-of-the-art mesoscale numerical weather prediction system (NWP) intended both for forecasting and atmospheric research. It is an Open Source project, created by a partnership of the US National Oceanic and Atmospheric Administration, the National Center for Atmospheric Research, and more than 150 other organizations and universities; it is used by a large fraction of weather and climate scientists worldwide. The WRF code base is written in Fortran-90 and is both complex and extensive (about a million lines of code). The governing equations set of the WRF model is based on the compressible, non-hydrostatic atmospheric motion with multiple physics processes, such as cloud and precipitation, boundary-layer turbulence, land–ocean–air interaction, radiative transfer in the atmosphere, and energy transfer at the surface. The finite difference method is used to discretize the governing equations of the WRF model. These discretized equations are integrated over time to obtain time-dependent atmospheric motion and physical states. Owing to the multiple physical processes that determine the atmospheric motion field, the number of the prognostic variables of the WRF model is quite large compared with a simple computational fluid dynamics model that consists of the Navier–Stokes equation and the mass continuity equation. The large number of three-dimensional prognostic variables is a severe computational constraint which requires high-performance computational resources.

In this paper, we focus on the advanced research version of WRF, called WRF-ARW (Advanced Research WRF) [7], which features very high resolution and is being used to explore ways of improving the accuracy of simulation of severe weather events, for example, tropical cyclones such as hurricanes and typhoons, tornadoes, windstorms, and heavy rainfall events. The WRF-ARW system components are depicted in Figure 1. The most computationally intensive components are the *Dynamics Solvers* and the *Physics Packages*.

## *1.2. Previous work on GPU acceleration of WRF*

Previous work in GPU acceleration of WRF is discussed in [9] by Michalakes, one of the main architects of WRF, and in [10]. Both papers deal with the WRF Single-Moment 5-class microphysics kernel,[§] which is one of the physics packages. Stand-alone GPU acceleration has also been examined for scalar advection terms in the context of WRF-CHEM (WRF model coupled with Chemistry). Because of the large amount of tracers needed in WRF-CHEM, the focus of this work was on overlapping CPU–GPU data transfers with computation.[¶] Mielikainen investigated the computational performance of GPU acceleration for a short-wave radiation scheme [11].

All these studies used experimental, stand-alone implementations and are not included in the standard WRF distribution. However, they demonstrate the potential for accelerating weather physics codes on GPUs.

[‡] http://www.wrf-model.org.

[§] http://www.mmm.ucar.edu/wrf/WG2/GPU/WSM5.htm.

[¶] This work has not been published in a peer-reviewed conference paper or journal but is available at http://www2.mmm.ucar.edu/wrf/WG2/GPU/Scalar_Advect.htm.

DOI: 10.1002/cpe

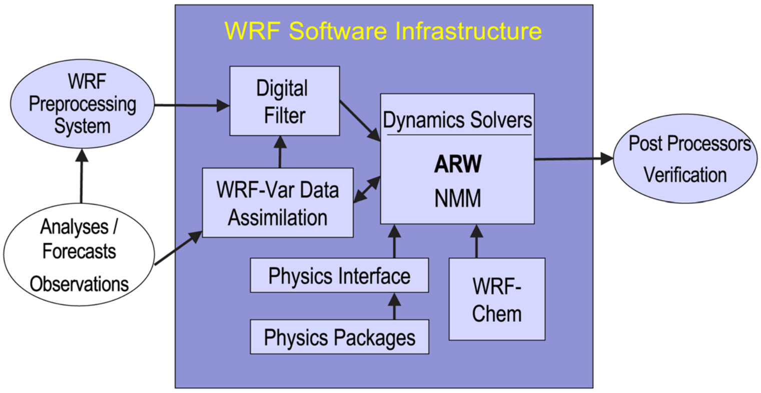

Figure 1. Weather Research and Forecasting-Advanced Research WRF (WRF-ARW) system components (from [7]).

Furthermore, all these implementations of accelerated kernels are in Compute Unified Device Architecture (CUDA). As CUDA is Nvidia's proprietary technology, usable only on Nvidia GPUs, we prefer to use OpenCL instead. OpenCL is an open standard and has the advantage that it can be deployed on GPUs, multicore CPUs, and other accelerators of different vendors. Recently, OpenCL support for FPGAs has become available; this is a very promising technology for NWP. Several other NWP codes have been adapted for GPU [12–14]. However, because of its size and complexity, a full GPU port of WRF has not yet been undertaken.

### *1.3. OpenCL programming*

*1.3.1. The OpenCL standard.* OpenCL [15] was developed by the Khronos Group in 2008 as an open standard for parallel programming of heterogeneous systems and is finding increasing adoption among providers of multicore CPUs and GPUs, (e.g., Nvidia, AMD, Intel, and ARM) and FPGAs (Altera). It provides an API for control and data transfer between the host and device (typically the host CPU and a GPU) and a language for kernel development. Contrary to proprietary solutions such as Nvidia's CUDA and Microsoft's DirectX, OpenCL is open and cross-platform, so that it can be deployed on different operating systems (Linux, OS X, and Windows) and hardware architectures (multicore CPUs, GPUs, and FPGAs). The OpenCL API is specified for C and a C++. In practice, the API is quite fine grained and verbose and requires many boilerplate codes to be written. Consequently, it is not straightforward to integrate OpenCL in existing codes, especially for non-computing scientists.

*1.3.2. Overview of the key OpenCL concepts.* In OpenCL parlance, the *Host* is the system that runs the top-level code, usually the CPU. The system that executes the parallel kernels is called the *Device*. Often, the Device is a GPU, but it can also be the same multicore CPU that acts as the Host, or a different accelerator such as an FPGA or an Intel MIC. The combination of the Host and one or more Devices is called the *Platform*. Furthermore, OpenCL uses the concepts of *Context*, *Command Queue*, *Buffer*, and *NDRange* to describe the movement of data between Host and Device, and the parallel execution of kernels on the Device. The Context is the environment within which kernels execute, in particular, it describes the memory accessible to the Device using Buffers, and the Command Queue used to schedule execution of a kernel or transfer of data. The NDRange (*N-dimensional range*) describes the parallel execution in terms of the number of compute units (cores) and hardware threads that will each run an instance of the kernel.

To compare OpenCL to OpenMP, the body of an OpenMP parallel for loop corresponds to the kernel, and the range of the loop corresponds to the NDRange.

DOI: 10.1002/cpe

### *1.4. The OclWrapper library*

To facilitate the integration of the OpenCL code into existing code bases, we developed the OclWrapper library,[‖] which supports C, C++, and Fortran-95. The library wraps the OpenCL Platform, Context, and Command Queue into a single object, with a much smaller number of calls required to run an OpenCL computation. As it is a thin wrapper, the additional abstraction comes at no cost in terms of features: the OpenCL API is completely accessible.

The OclWrapper library consists of several components:

**The OclWrapper C++ class**. This class abstracts the OpenCL concepts of Platform, Context, Device, and Command Queue into a single object. Using C++ features such as templates, polymorphic functions, and default arguments, it provides a greatly simplified interface that is suitable for the majority of OpenCL applications. However, as the class instantiates all the lower-level OpenCL objects, all low-level OpenCL features are still accessible without overhead.

**The oclWrapper Fortran-95 Library**. This library provides the *oclWrapper* Fortran module, which offers a subroutine-based interface to the C++ OclWrapper class. As Fortran does not offer polymorphic subroutines, the library provides individual functions for manipulating multi-dimensional arrays of various types.

**The oclBuilder SCons Library**. To build the OclWrapper, we use the SCons[**] build system, a replacement for Make that allows the writing of very complex build scripts in Python. The oclBuilder library makes it possible to write a build script for our OclWrapper in a few lines.

The use of the library is illustrated subsequently on a simple C++ OpenCL example.

```
// Create wrapper for default device and single kernel
OclWrapper ocl(srcfilename,kernelname,opts);
// Create read and write buffers
cl::Buffer rbuf = ocl.makeReadBuffer(sz);
cl::Buffer wbuf = ocl.makeWriteBuffer(sz);
// Transfer input data to device
ocl.writeBuffer(rbuf,sz,warray);
// Set up index space
ocl.enqueueNDRange(globalrange, localrange);
// Run kernel
ocl.runKernel(wbuf,rbuf ).wait();
// Read output data from device
ocl.readBuffer(wbuf,sz,rarray);
```

Without the wrapper, the same program would be about a hundred lines of code, and each individual call would have many arguments.

The use of the wrapper in Fortran is equally straightforward:

```
use oclWrapper
integer(8) :: rbuf, wbuf
real :: dimension (ims:ime,kms:kme,jms:jme) rarray
real :: dimension (ims:ime,kms:kme,jms:jme) warray
integer :: globalrange, localrange
! Create wrapper for default device and single kernel
call oclInit(srcfilename,kernelname)
! Create read and write buffers
call oclMake3DFloatArrayReadBuffer(rbuf,sz,rarray)
call oclMake3DFloatArrayWriteBuffer(wbuf,sz)
! Transfer input data to device
call oclWrite3DFloatArrayBuffer(rbuf,sz,rarray)
! Run kernel over index space
oclRun(globalrange, localrange)
! Read output data from device
oclRead3DFloatArrayBuffer(wbuf,sz,warray)
```

[‖] https://github.com/wimvanderbauwhede/OpenCLIntegration.

[**] http://scons.org/.

Table I. Specifications of hardware platforms used in this work.

| | # of cores | Vector size | Clock speed (GHz) | CPI | Memory BW (GB/s) |
|---|---|---|---|---|---|
| CPU: Intel Xeon E5-2640 | 24 | 8 | 2.5 | 480 | 42.6 |
| GPU: Nvidia GeForce GX480 | 15 | 32 | 1.4 | 672 | 177.4 |
| CPU: AMD Opteron 6176 SE | 48 | 4 | 2.3 | 441.6 | 42.7 |
| CPU: AMD Opteron 6174 | 24 | 4 | 2.1 | 201.6 | 42.7 |
| GPU: Nvidia Tesla C2070 | 14 | 32 | 1.1 | 492.8 | 144 |

Crucially, the Fortran oclWrapper is implemented as a Fortran-95 module which stores the OclWrapper object globally, so that the API subroutine calls can be issued in different program units. This is an essential feature for integration into an existing code base.

### *1.5. Hardware performance indicators*

*1.5.1. Hardware platforms used in this work.* To evaluate our work, we used several different systems. The host system used for the main experiments was based on an Intel Xeon E5-2640 CPU (dual-processor, 6 cores/chip, 2 threads/core). This processor has a 256-bit AVX SIMD, so a smart compiler will generate code capable of performing up to 8 floating point operations in parallel. The GPU was an Nvidia GeForce GX480. It has 15 compute units with 32 processing elements each.

Two other systems, a 48-core AMD Opteron 6176-SE system (four 12-core processors) and a Tesla C2070 GPU hosted on a 24-core AMD Opteron 6174 system (two 12-core processors), were used for additional experiments (see Table I for full details).

*1.5.2. Computational performance indicator.* We define the (single-precision floating point) computational performance indicator *CPI* as

```
CPI= #threads × SIMD width × clock freq
```

This figure is directly proportional to FLOPS but more easy to obtain. We define 'threads' as the product of the number of cores/compute units and their hyperthreading capability, and 'vector size' as either the SIMD vector size or the number of processing elements per compute unit. The CPIs for our platforms are shown in Table I.

From Table I we see that purely in terms of computation, under optimal circumstances, the GeForce GPU can be at best 1.4× faster than the Intel CPU; the CPI of the Tesla GPU is only 3% higher than that of the Intel CPU. If the memory bandwidth is the limiting factor, the achievable speed-up for the application running on the GPU would be 4.2×.

*1.5.3. Communication bandwidth limit on achievable performance.* The total achievable speed-up is limited by the data transfer rate between host memory and GPU memory, and the overhead for control of the GPU. According to our measurements our system can achieve about 2 GB/s reading from the GPU and 8 GB/s writing to the GPU. In Section 5, we present the detailed discussion of the cost of data transfer and computation. In general, we can analyze the achievable speed-up as a function of the computational speed-up (which in its term depends on the CPI and the memory bandwidth) and the data transfer speed. Figure 2 shows a generic graph which can be used to assess the performance of an algorithm.

What the graph shows is the achievable speed-up as a function of the CPU compute time relative to the data transfer time, with the GPU/CPU computational speed-up as a parameter. For example, if the computation on the CPU takes 100 ms and the data transfer 1000 ms, then there can be no speed-up, no matter how fast the GPU computes. On the other hand, if the CPU takes 1000 ms and the transfer time is 100 ms, then with a GPU/CPU computational speed-up of 5× the total speed-up $= 1000/(100 + 1000/5) = 3.3\times$. In the legend, 'pipelined' means that the computations and data transfers overlap in pipelined fashion, which can improve performance when processing a stream of data, and if the transfer and compute times are of the same order. The formulas for the non-pipelined and pipelined cases are in general:

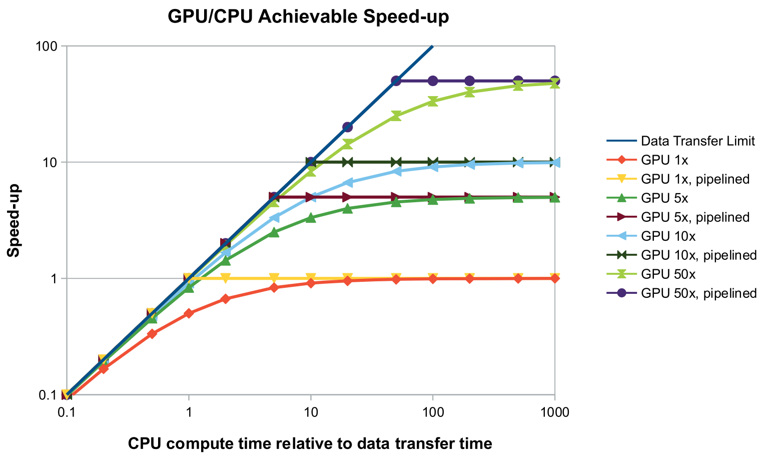

Figure 2. Achievable speed-up from offloading work to the graphics processing unit (GPU).

$$s_{non\text{-}pipelined} = \frac{\alpha}{1 + \frac{\alpha}{\beta}} \tag{1}$$

$$s_{pipelined} = \frac{\alpha}{max\left(1, \frac{\alpha}{\beta}\right)} \tag{2}$$

where

$$\alpha = \Delta t_{CPU} / \Delta t_{transfer,GPU}$$
$$\beta = \Delta t_{CPU} / \Delta t_{compute,GPU}$$

## 2. METHODOLOGY

To assess the feasibility of GPU acceleration of WRF, we used the following approach:

1. Single-node performance evaluation of the current WRF software using Message Passing Interface (MPI) and OpenMP
2. Profiling of WRF runs with a number of different configurations
3. Selection of code portions suitable for acceleration via data-parallel computation
4. Implementation of the code in OpenCL
5. Performance evaluation of the OpenCL kernel
6. Integration of the OpenCL kernel into the WRF code

## 3. WRF-ARW PERFORMANCE ANALYSIS

### 3.1. *Settings of the WRF simulation*

The version of the WRF-ARW model used here is version 3.4, released in April 2012. The fifth-order upwind-biased scheme is used for the discretization of the advection terms in the horizontal direction; the third-order upwind-biased scheme is used for the discretization of the advection terms in the vertical direction, and the third-order Runge–Kutta scheme is used for the time integration of the governing equations [7].

The case investigated in the present numerical simulations is the severe tornado case that occurred in Tsukuba, Japan, a suburban area north of the Tokyo metropolitan region, on 6 May 2012. This tornado caused severe damages in Tsukuba and its surroundings and was rated as F3 on the Fujita tornado damage scale. The simulation was set up with the full physics modules implemented in

DOI: 10.1002/cpe

WRF. Because the meteorological case chosen here is a tornado that was generated by a well-developed cumulonimbus cloud system, one of the most important physical processes is the cloud and precipitation process, a so-called microphysics process. The WRF Single-Moment 6-class microphysics module [16] is used for the microphysics parameterization, because this scheme is one of the most sophisticated single-moment schemes and is successful in dealing with convective storms in moist regions such as East Asia [16].

The WRF model allows nesting of multiple computational domains in a larger domain. The present study explores the computational performance of two cases of domain settings: one is a single domain, and the other case uses triple-nested domains. For the single domain case, the horizontal grid spacing is 5 km, while for the triple domain case, the grid spacings are 2.5 km, 500 m, and 100 m.

### 3.2. *MPI versus OpenMP on a single compute node*

The focus of our work is on acceleration of code running on a single compute node through offloading of parts of the code to an accelerator such as a GPU. Therefore, in this work, we deployed WRF on a single node consisting of a multicore CPU and GPGPU. We did not use a cluster of nodes in our experiment, as the scalability and performance of WRF in a cluster is determined by the MPI subsystem, and our GPU/multicore acceleration approach is entirely orthogonal to this. In other words, from a cluster perspective, an accelerated node is simply a faster node, and the MPI performance and scalability of WRF has already been investigated in detail on a variety of systems [17–19].

To establish the baseline performance, we carried out a number of experiments of WRF runs with MPI and OpenMP. The first sets of results (Figure 3) was obtained on the AMD 24-core system, using the GNU Fortran compiler (*gfortran v4.4*). It compares OpenMP with MPI performance for a small domain size without nesting:

```
    e_we = 100,
    e_sn = 100,
    e_vert = 27,
```

where $e_{we}$, $w_{sn}$, and $e_{vert}$ are the sizes of computational grid in the east–west, the north–south, and the vertical direction. The second set of results is obtained on the 48-core AMD system, using the Intel Fortran compiler (*ifort*). A third set of results was obtained on the 12-core (24-thread) Intel Xeon system; it shows an OpenMP performance for varying numbers of threads. Both the second and third experiment simulate a larger domain with nesting:

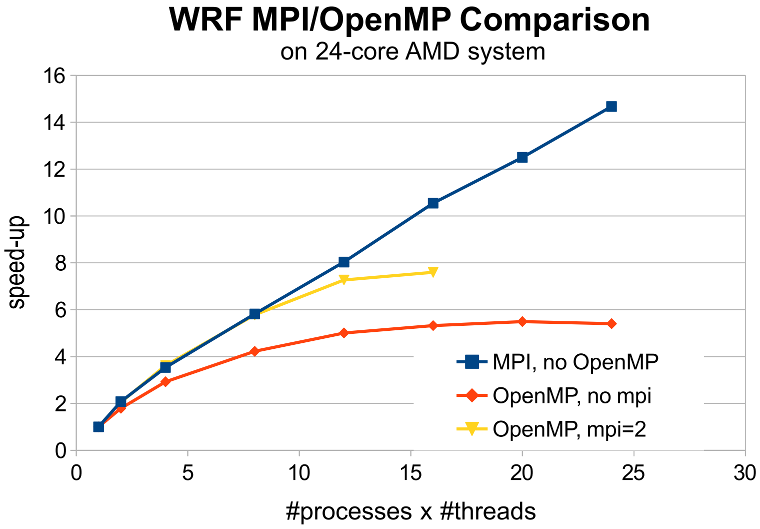

Figure 3. Weather Research and Forecasting (WRF) performance with Message Passing Interface (MPI) and OpenMP, domain size 100 × 100 × 27, on the 24-core AMD system.

DOI: 10.1002/cpe

```
e_we = 500,301, 501,
e_sn = 500, 301, 501,
e_vert = 60, 60, 60
```

The other WRF settings are identical. The second and third sets are shown in Figure 4.

The figures show the speed-up as a function of the number of *parallel processes* × *threads*. Missing points indicate that the simulation failed to complete.

The conclusions from these experiments are clear:

- First, performance varies considerably across platforms.
- Second, for large domains, the speed-up saturates at about half the number of physical threads.
- Finally, MPI outperforms or matches OpenMP for all cases.

These findings are in line with other studies, for example, [20]. The poor OpenMP performance is due to the sub-optimal use of OpenMP in WRF: the use of many shared variables results in frequent locking. This is a result of the decision not to rewrite the code for OpenMP, but to rely only on insertion of pragmas.

The observed behavior constitutes a problem for GPU acceleration and would need to be addressed: in current systems, the GPU can only be accessed by a single process at a time. As MPI creates separate processes, access to the GPU would be serialized. Furthermore, each computation on the GPU would be on the portion of the total memory space used by the MPI process, rather than on the full memory space. So either all processes would have to copy their memory space to the process controlling the GPU, or the GPU would have to be called sequentially by each process in turn. Either way, as a result, the overhead of accessing the GPU would dominate the performance, and the net result would be a slow-down rather than a speed-up. With effective use of OpenMP, it should be possible to match or even better the MPI performance, as intrinsically, OpenMP has a lower overhead. The host code would be a single process with a single memory space and could easily and effectively interact with the GPU. An alternative solution would be to rewrite the MPI code to include an additional process which would have access to the combined memory space and control the GPU. As modern operating systems uses copy-on-write, this approach should also result in good performance and might even be preferred.

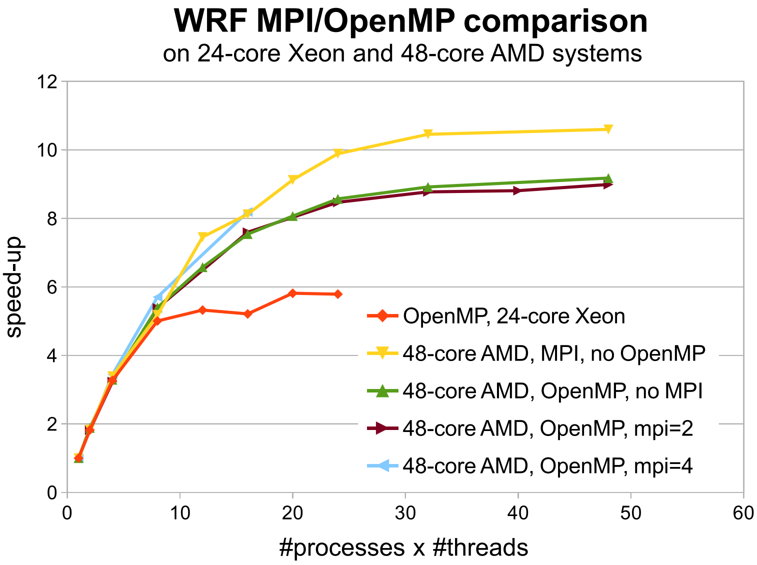

Figure 4. Weather Research and Forecasting (WRF) performance with Message Passing Interface (MPI) and OpenMP, domain size 500 × 500 × 60 with nesting, on the 24-core Intel and 48-core AMD systems.

Table II. Contributions of various parts of Weather Research and Forecasting (WRF) to total run time (%).

| functionality | modules | test1 | test2 |
|---|---|---|---|
| dynamics | | 71 | 65 |
| | advection:<br>advect_scalar<br>advect_scalar_pd | 28 | 20 |
| | small-step | 17 | 17 |
| | big-step | 17 | 18 |
| | other | 9 | 10 |
| physics | | 14 | 21 |
| | microphysics | 7 | 12 |
| | other | 7 | 9 |

### 3.3. WRF run profiling

We profiled the two typical WRF runs (on a 256 × 256 × 32 domain) using a sampling profiler (Shark on OS X 10.6.8). The conclusion of these experiments was that most of the time is spent in the dynamic core and, to a lesser extent, the physics modules (Table II).

Together, dynamics and physics constitute about 85% of the total run time. This time is divided across a large number of calls to different routines, so there is no 'quick win'. However, the contributions of advection, big-step and small-step routines, and microphysics already account for 70% of the total run time, so these parts of the model constitute a logical focus for acceleration. These findings are in line with those for the COSMO model [13]. Note that accelerating 70% of the code 5× results in a speed-up of 2.3×, a speed-up of 10× on 85% of the code would result in 4.3×. Furthermore, the structure of the various dynamics kernels is similar in terms of the required approach to parallelization, so that by studying one of the kernels, we can infer the behavior of the other kernels.

## 4. OPENCL KERNEL FOR SCALAR ADVECTION

As can be seen from Table II, a relatively large part of the run time for WRF is spent in the scalar advection routines *advect_scalar* and *advect_scalar_pd*. These routines are part of the dynamic core (*dyn_em*), and no previous GPU implementations have been reported in the literature. As all the advection routines are all similar in structure, the OpenCL version of *advect_scalar* can serve as a template for the other routines.

### 4.1. Approach to parallelization and OpenCL porting

The original WRF kernel for scalar advection consists of a number of nested loops over $i$,$j$, and $k$, where typically, the inner loops are guarded by *if*-statements. Also, the code uses arrays to store all intermediate results. First, we translated the code to C, using *F2C_ACC* [12]. We then analyzed the conditionals and replaced all run-time *if*-statements that are actually run-time constants with preprocessor *#if*-statements. This is important for GPU kernels, as run-time branching of a thread in a single warp will lead to stalling of the threads that do not execute the selected branch. The next step was an analysis of the loop structures and boundaries, resulting eventually in a single, unified nested loop with conditionals inside.

This approach is not appropriate for single-threaded code, as the new code executes more statements because the conditions are evaluated for every combined loop iteration. However, for data-parallel execution, the placement of the conditionals as in the original code would not result in reduced run times, only in thread stalling.

We then replaced the intermediate arrays with local variables and removed some loop dependencies by computing 'ahead of time'. Finally, we merged the loops into a single loop and then used the range of this loop as the index space (the global *NDRange*). The local NDRange was set to the

DOI: 10.1002/cpe

*k*-range. We experimented with different approaches and values for global and local ranges but found that the aforementioned configuration was optimal.

In terms of effort, the total elapsed time to parallelize the kernel, port it to OpenCL, and validate it was about 1 month, for an experienced computing scientist. The total project including the software engineering required to integrate the OpenCL code seamlessly into the original WRF code took 2 months.

### 4.2. Implementation

The structure of the kernel is shown in Algorithm 1. The array *ranges_boundaries_degrade* contains the various ranges (*ims*, *ime*, etc.) and computed boundaries (*i_start, i_end*, etc.) and conditions (*degrade_xs*,...) for the computation. It is more efficient to pass these to the kernel as an array than as individual arguments. The *zero_tendency* argument is used to determine the part of the kernel to be executed (first zero the *tendency* array then compute the new values). Every thread only uses a small portion of the *field* array (typically ±3 grid points in every dimension), therefore, we copy the required values to local arrays for $i, j$, and $k$. The functions *calc_tendency_** contain the advection computations for the x, y and z dimensions.

**Algorithm 1** Structure of the OpenCL scalar advection kernel

```
__kernel void advect_scalar (
  __global float *tendency,
  __global const float *field,
// ... other data ...
  __constant const int *ranges_boundaries_degrade,
  __constant const int *zero_tendency
) {
  int gl_id = get_global_id(0);
  if (zero_tendency[0]!=0) {
  // zero the tendency array
    tendency[gl_id]=0.0;
  } else {
  // assign ranges_boundaries_degrade to local variables for convenience
  // calculate the ranges for i, j, k
  // calculate i,j,k from the global index
  // create a local copy lfield of the field entries for i,j,k needed for calculating the fluxes
  // read tendencies for x, y and z
    float tend_ikj=0.0;
  // calculate the tendencies for x, y and z
    if (j>=j_start_y && j<=j_end_y) {
      tend_ikj = calc_tendency_y_l(...);
    }
    if (i>=i_start_x && i<=i_end_x) {
      tend_ikj = calc_tendency_x_l(...);
    }
    tend_ikj= calc_tendency_z_l(...);
  // write the result to main memory
    tendency[...] = tend_ikj;
  }
}
```

### 4.3. Verification

In order to verify that our OpenCL kernel code produces the same results as the original Fortran code, we employed a testing approach where both codes are run in succession on identical input

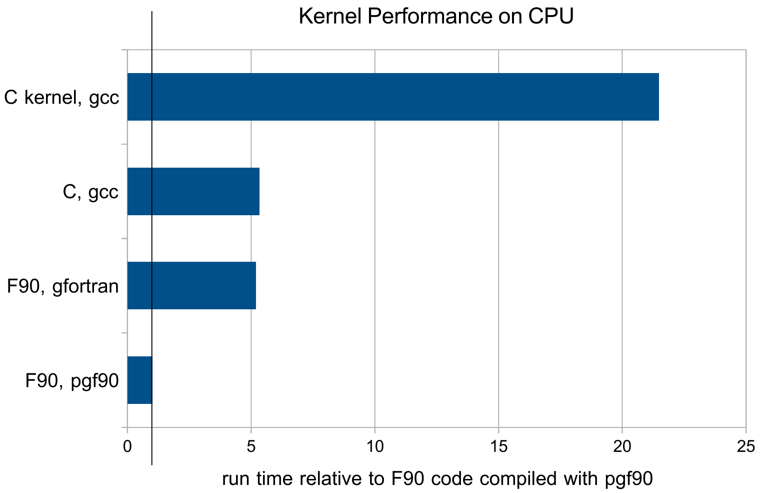

Figure 5. Kernel performance compared with original code, both running single-threaded on CPU.

values, and the results computed by each are compared at run time using a set of comparison functions. To account for differences in rounding errors arising from the different order in which the floating point instructions are executed and different rounding algorithms implemented on the CPU and GPU, we allowed an error of $5.10^{-6}$. This value is ad hoc but based on tests with simple floating point arithmetic kernels.

### *4.4. Performance evaluation*

The rewritten code is intended for data-parallel execution on a GPU. It is therefore expected that the code will run slower than the original sequential code, which was optimized for single-threaded execution. This is confirmed by our measurements (Figure 5): the data-parallel kernel, when run sequentially, is about 4× slower than the original Fortran code compiled with *gfortran* or the equivalent C code compiled with gcc, with optimization *-O3*. This is expected because, as detailed in Section 4.1, the rewritten code executes many more instructions than the original code, as a result of moving conditional branches from the outside to the inside of loops. However, when compiled with the PGI Fortran compiler *pgfortran*, with optimizations *-fast -fastsse -Mipa=fast*, the original Fortran code is 5× faster compared with the *gfortran* binary. This is a result of the better vectorization performance of the PGI compiler: the Xeon E5-2640 CPU has 256-bit AVX vectors, so it can in principle handle 8 single-precision floating-point operations in parallel. The gcc compiler does not vectorize the code, hence the observed performance difference. We also tested the effect of the auto-parallelization option *-Mconcur* with various sub-options, but this did not result in performance improvement.

Figure 6 shows the performance of the actual OpenCL kernel (which is essentially the same code as the C kernel but parallelized using the OpenCL framework), relative to the performance of gcc/gfortran, which we chose as the reference because it is available on all CPU platforms we used. We observe a speed-up of about 12× on the GeForce GPU (the reasons for the lower performance on the Tesla GPU are discussed in Section 5).

We also evaluated the performance when parallelizing the kernel execution with OpenMP (Figure 7). On the AMD system, the speed-up for 24 threads was 21×. The performance on the Intel CPU was also very good but saturated at 8× at 12 threads, this shows that the performance of a hyperthreaded core is less good than that of two separate cores for this type of code. This is because all threads are busy most of the time: hyperthreading works essentially by allowing more than one thread to run per core, but this mechanism is only effective when the threads are stalling a lot of the time: under such circumstances, without hyperthreading, the CPU would idle; with hyperthreading, the CPU is used by one thread while the other is stalled.

DOI: 10.1002/cpe

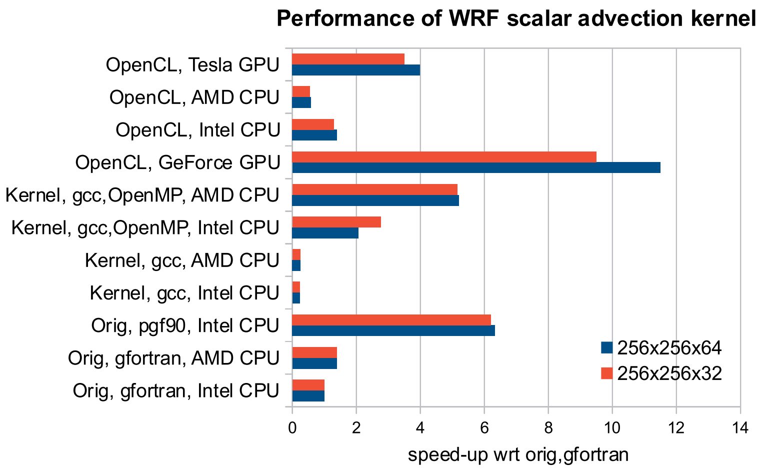

Figure 6. OpenCL kernel performance.

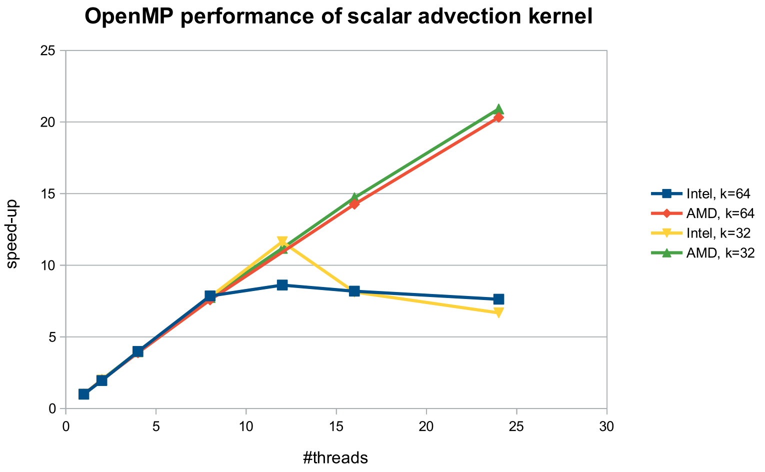

Figure 7. OpenMP performance of the scalar advection kernel, $(i, j, k) = (256, 256, 32)$ and (256,256,64).

### 4.5. *Discussion of kernel performance*

At first sight, it might seem from these results that the GPU acceleration is hardly worthwhile: the OpenCL code deployed on the GeForce GTX 480 GPU is only about twice as fast as the original code when compiled with the PGI compiler. However, it is important to realize that accelerating only the scalar advection kernel would not speed up WRF execution anyway, as it accounts for only about 10% of the run time.

As explained in Section 3.3, a large portion of the code base must be accelerated to the GPU to achieve considerable speed-ups of the total application (Amdahl's law). As we will see from the analysis in Section 5, under those circumstances, GPU acceleration can result in a considerable performance increase.

## 5. GPU RUN TIME ANALYSIS

The specifications in Section 1.5 provide a good guideline for the achievable performance; however, to obtain a clear picture, in this section we present an analysis of the performance of the *advect_scalar* kernel on a GeForce GTX480 and a Tesla C2070 GPU.

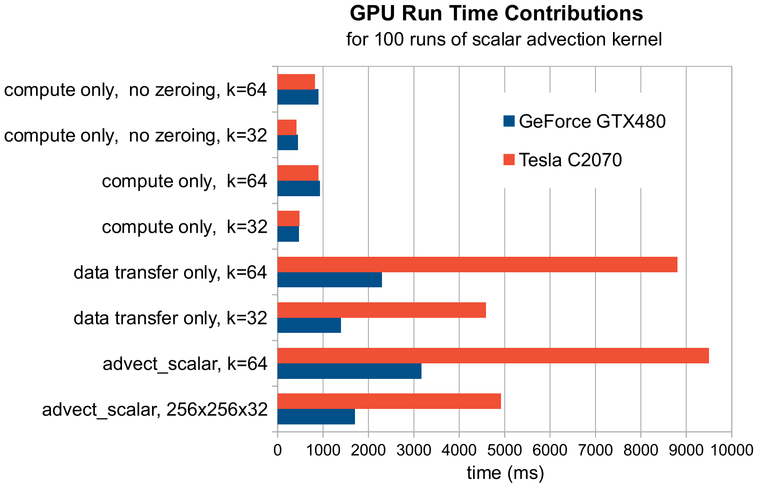

Figure 8. Run time contributions for GeForce GTX 480 graphics processing unit (GPU) on Intel host and Tesla C2070 on AMD host.

### *5.1. Experiments*

We performed the following experiment on the advect_scalar kernel, with a domain size of 256 × 256 × 64:

- For each run of the GPU, the host:
  - writes 4 buffers of the domain size (16~MB) to the GPU memory and a number of smaller buffers, transferring a total of about 64.25 MB
  - calls the GPU twice: first to zero the tendency array, then to compute the new tendencies
  - reads the new tendency array, 16 MB
- The host performed 100 runs in a loop and recorded the aggregate run time.
- This experiment was repeated 20 times

The run time contributions for both GPUs are shown in Figure 8. In this figure, 'compute only' means no data transfer from host memory to GPU memory; 'data transfer only' means that the kernel is called but performs no computation; and 'no zeroing' means that the call to zero the tendency array is skipped, so the GPU is called only once per run.

We also investigated the influence of the data size on the performance. The experiment was the same as earlier, but we varied the domain size as follows:

```
i,j: 32,64,128,256,512,1024
k: 32,64
```

Figure 9 shows the speed-up of the OpenCL GPU code (including data transfers) compared with the original code with the GNU and PGI fortran compilers.

There are several interesting points about these results:

*5.1.1. Influence of host system on transfer time*. The first is the difference in transfer time: the AMD/Tesla system takes almost 4× longer than the Intel/GeForce system: the transfer bandwidth for the Intel/GeForce system is 2.8 GB/s (100 transfers of 64 MB in 2.3 s), which is reasonably close to the top performance of 4 GB/s for a 16-lane PCI express with 2.5 GT/s transfer rate; however, the AMD/Tesla system only reaches 730 MB/s. Looking closer at the PCIe specs of both systems, the only difference is the latency: both systems have a 16-lane PCI Express v2, 2.5 GT/s, but the more recent Intel system has a latency less than 256 ns, whereas the AMD system has a latency less than 1 μs. The AMD system also has a significantly lower memory bandwidth (Figure 10). Another

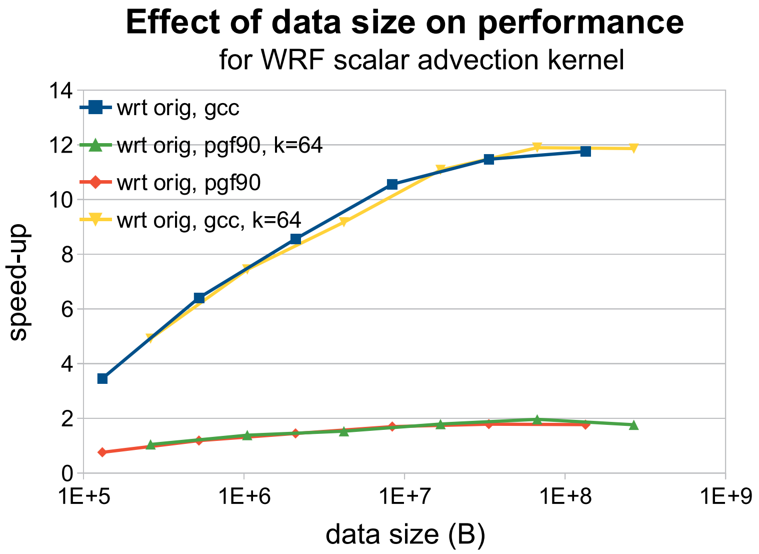

Figure 9. Influence of data size on performance. WRF, Weather Research and Forecasting.

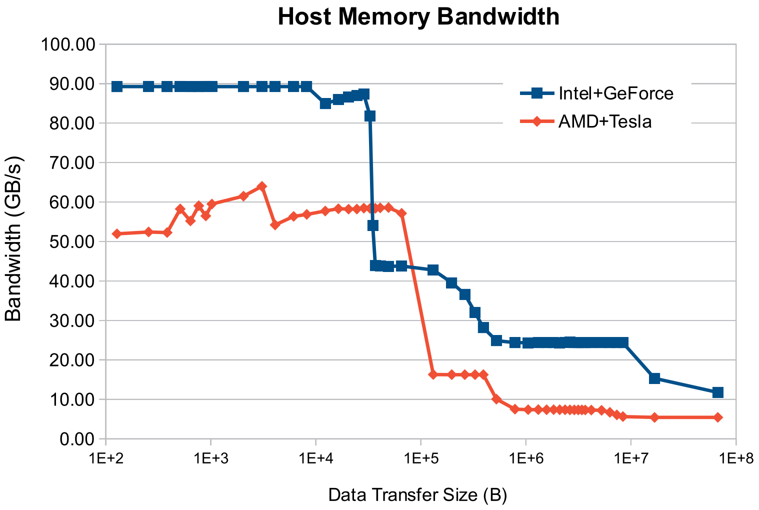

Figure 10. Host memory bandwidth.

factor that most likely influences the transfer time is the smaller cache size; in any case, for a transfer size of greater than 1 MB, the memory bandwidth of the AMD CPU is 2.5× lower than that of the Intel CPU.

It should be noted that a more modern system with PCIe v3 is capable of 8 GT/s (with a more efficient encoding), so the achievable performance of the GPU computation would be considerably better: for the scalar advection kernel, the total run time would be reduced by a factor of two.

*5.1.2. GPU compute performance.* The second observation is that the Tesla GPU computes the kernel about as fast as the GeForce. This is not really surprising when comparing the specs of both GPU cards: the main difference is in the amount of on-board memory. Nvidia mentions that the floating point performance of the Tesla cards is better than that of the ‘consumer cards’, but that applies only to a double-precision floating point. As the WRF uses single precision, the much cheaper GeForce card is the better choice.

DOI: 10.1002/cpe

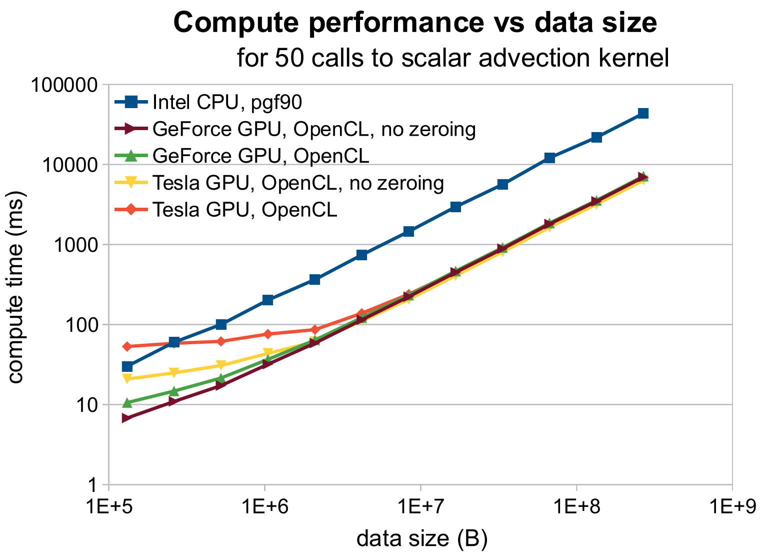

Figure 11. Compute performance comparison of original code on Intel CPU, compiled with the PGI Fortran compiler, to OpenCL code on Tesla and GeForce graphics processing units (GPUs).

*5.1.3. Influence of the data size.* We see from Figure 9 that the speed-up of the GPU increases for larger data sizes. The reasons for this behavior are twofold: on the one hand, the fixed cost for starting the GPU is relatively less important for larger data transfers. On the other hand, the computation grows more than linear with data size, so the cost of the data transfer is less dominant for larger data sizes. There are other factors, for example, cache misalignment is less important on larger transfers. All these factors contribute to the observed behavior.

If we compare *only* the compute performance of the GPU with the original code on the Intel CPU (compiled with the PGI compiler), we see that the GPU is up to 7× faster (Figure 11).

*5.1.4. Comparison to the achievable performance.* The results in Figures 6, 9, and 11 can be related to the generic achievable performance graph (Figure 2) discussed in Section 1.5.3: from Figure 11, the 'GPU/CPU computational speed-up' (the parameter for the set of curves in the graph) is 7×; the 'CPU compute time relative to data transfer time' (X-axis of the graph) is the run time on the CPU for code compiled with the PGI compiler (5760 ms[††] for the largest data size, 64 MB) divided by the corresponding data transfer time (2300 ms from Figure 8). For the GeForce GPU, this results in a factor of 2.5. Using this value, we can compare the achieved speed-up (2×; Figure 9) against the theoretically achievable speed-up from Figure 2, and we see a close correspondence: the performance is dominated by the data transfer, and the achievable speed-up is indeed limited to very nearly a factor of two (the theoretically computed value using Equation 1 is 1.85). This exercise demonstrates the usefulness of our theoretical model to estimate achievable performance.

*5.1.5. Conclusions of the run time analysis.* In summary, the conclusions of the GPU performance analysis are the following:

- The host-GPU link is the main bottleneck, and care must be taken in the choice of the host platform, in particular memory bandwidth and PCIe latency.
- The GPU needs to work on large data sizes for optimal performance. For our kernel, performance is optimal for data sizes $>$ 64 MB.
- If the complete computation was performed on the GPU, this would yield a speed-up of 5× – 10× compared with the original code compiled with the PGI compiler.

[††] This value is not present *as-is* in the figures but can be computed from Figures 6 and 8.

DOI: 10.1002/cpe

## 6. INTEGRATION OF THE OPENCL CODE INTO WRF

As part of this research, we developed a strategy for integrating OpenCL code into large Fortran codebases such as WRF. We created a Fortran OpenCL wrapper library to facilitate the integration, as discussed in Section 1.4.

### *6.1. Overview of WRF code structure*

The WRF code structure for the dynamic kernel, omitting details, is shown in Figure 12. To integrate the OpenCL code in the host code, we added a single *use* statement and a single call to *wrf_init* in *main/module_wrf_top.F*:

```
subroutine wrf_init( no_init1 )
! ... other use statements ...
use module_init_ocl
! ... original wrf_init code ...
call advect_scalar_init_ocl_grid ( head_grid )
end subroutine wrf_init
```

This is the only change to the initialization code.

The change to the dynamic kernel code is also minimal: in the subroutine *rk_tendency* (in *dyn_em/module_em.F*) we added

```
use module_advect_scalar_ocl
```

and we replaced the call to *advect_scalar* using the preprocessor by a call to *advect_scalar_ocl*:

```
call advect_scalar_ocl
```

We could have given the OpenCL routine the same signature as the original *advect_scalar* routine, but for clarity, we prefer to have a separate name indicating that the routine is an OpenCL routine.

The actual *new* source code is entirely contained in two new modules, *init_ocl* and *advect_scalar_ocl*.

The first module contains two subroutines, *advect_scalar_init_ocl_grid* and *advect_scalar_init_*ocl. The first routine (Algorithm 2) is called in *wrf_init* as shown earlier. It is essentially a wrapper routine which extracts information from the *head_grid* datastructure and passes it on to *advect_scalar_init_ocl* (Algorithm 3), which performs the OpenCL framework initialization. The main actions in this routine are loading and compiling the kernel, creating the buffers and setting the kernel arguments.

The second module (Algorithm 4) runs the OpenCL scalar advection kernel on the GPU. Its main actions are writing the data to the GPU, running the GPU, and reading back the data. Note that the GPU is run twice, once to zero the tendencies and once to compute the new tendencies.

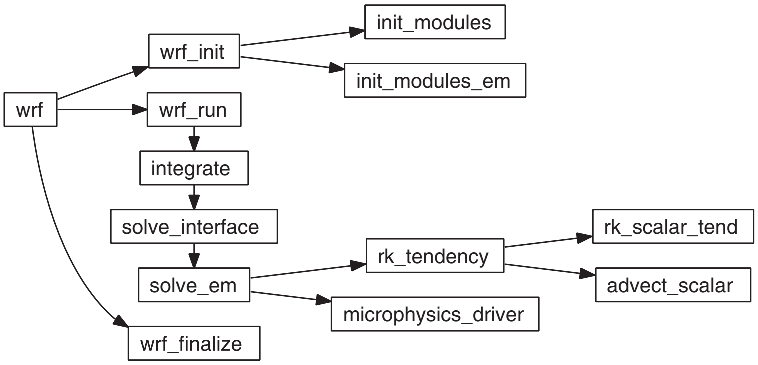

Figure 12. Weather Research and Forecasting (WRF) code structure.

DOI: 10.1002/cpe

**Algorithm 2** OpenCL initialization wrapper to extract info from grid

```
subroutine advect_scalar_init_ocl_grid ( grid )
    use module_domain
! Variable declarations
...
    call get_ijk_from_grid (  ... )
! Loop range computations
...
    call nl_get_time_step ( 1, time_step )
! Call actual OpenCL initialization routine
    call advect_scalar_init_ocl ( ... )
end subroutine
```

**Algorithm 3** OpenCL initialization routine

```
subroutine advect_scalar_init_ocl (...)
  use oclWrapper
! Load the kernel source and compile for the platform
  srcstr='advect_scalar_ocl.cc'
  kstr='advect_scalar'
  call oclInit(srcstr,kstr)
! Set up the ranges
  oclGlobalRange = gl_range
  oclLocalRange = 0 ! NullRange
! Create the buffers
  call oclMakeWriteBuffer(tendency_buf,jikmfsz)
  call oclMakeFloatArrayReadBuffer(field_buf,jik_sz,field)
  !...
! Set the Kernel arguments
  call oclSetFloatArrayArg(0, tendency_buf )
  !...
! Assign to module array for convenience
  oclBuffers(1) = tendency_buf
  !...
end subroutine ! advect_scalar_init_ocl
```

**Algorithm 4** OpenCL driver code for scalar advection kernel

```
subroutine advect_scalar_ocl (...)
  use oclWrapper
  !...
! Write buffers to GPU memory
  call oclWriteBuffer(field_buf,jik_sz,field);
  !...
! First zero the tendency array on the GPU
  zero_tend(1)=1
  call oclWriteIntBuffer(zero_tend_buf,zero_tend_sz, zero_tend);
  call runOcl(jikmsz,0)
! Then compute the new tendencies
  zero_tend(1)=0
  call oclWriteIntBuffer(zero_tend_buf,zero_tend_sz, zero_tend);
  call runOcl(oclGlobalRange,oclLocalRange)
! Read back results from GPU
  call oclReadBuffer(tendency_buf,jik_sz,tendency)
end subroutine advect_scalar_ocl
```

To extend this work, rather than making separate calls to all the different dynamics and physics routines, the aim is to create an OpenCL version of the full *solve_em* subroutine. Then we can simply replace *solve_em* by *solve_em_ocl* in quite the same way as earlier.

## 7. DISCUSSION

The main research questions we set out to answer in this work were as follows: Is hardware acceleration of the Weather Research and Forecasting model on GPUs feasible and worthwhile?

First, we studied the code and performed experiments on the current parallel performance using MPI and OpenMP. We found that the WRF OpenMP performance is sub-optimal as it performs worse than MPI, whereas in principle, OpenMP should have a considerably smaller overhead. We analyzed the reasons for this behavior and concluded that to amend it is a major effort. We also observed that the MPI behavior is strongly sub-linear and saturates typically when 50% of the available hardware threads have been used, and at a performance of less than half the maximally achievable performance. In other words, it is in principle possible to speed up WRF considerably.

Then we profiled WRF runs to identify the most important routines in terms of run time. Our findings, confirmed by other authors, are that the dynamics and physics account for the majority of the WRF run time. As there has been previous work on acceleration of physics modules, we focused on the dynamics, and in particular, we chose the scalar advection module as the target for our study, as it is the dominant routine in terms of run time. By implementing the GPU kernel and evaluating its performance, we obtain more detailed answers to the questions of feasibility and pay-off.

Before discussing the acceleration of the WRF scalar advection kernel, we want to discuss the capabilities of the hardware platforms used in this work. There are many unrealistic expectations considering the achievable performance of multicore CPUs and GPUs. To help understand the performance of these systems, we defined indicators for the compute capability and memory bandwidth. From these indicators, we concluded that for computation-dominated code, the theoretical speed-up achievable by running the code on the GPU is quite small: moving the code from the Intel Xeon to the Nvidia Geforce could result in a speed-up of 1.4×; moving the code from the AMD host CPU to the Nvidia Tesla C2070 can at best provide a 3% speed-up. As noted earlier, the reason for this smaller improvement is the higher transfer cost on the AMD system. Of course, these indicators ignore the effect of the implementation of the code and the compiler performance, but they give an indication of what is achievable in terms of the hardware capability. What this means is that if one achieves a higher speed-up than these figures, either the application is not computation dominated, or the coding is sub-optimal, or the compilation is sub-optimal. If the code is memory bandwidth dominated, we see that the achievable speed-up is about 4×.

The approach to parallelization of the advection kernel using OpenCL is discussed in detail in Section 4.1; the conclusion is that it is definitely feasible and can result in a very good performance. However, we want to focus on the findings from the performance evaluation. In our opinion, the most important finding is that, in order to achieve the best possible performance on either a multicore CPU or a GPU, it is necessary to considerably rewrite the code for data-parallel execution. The other key finding is that the current system architecture is problematic for GPU acceleration, because the PCIe bus performance constitutes a huge bottleneck. However, it is still possible to achieve good performance provided that a substantial part of the model code is implemented on the GPU.

On the other hand, one has to ask the question if it is at all worthwhile to offload the code to the GPU. To evaluate this question, we used both OpenCL and OpenMP to parallelize our kernel code on the multicore CPU. As argued in the preceding discussion, there is in fact theoretically almost no difference in performance between the Intel Xeon E5-2640 multicore CPU and the Nvidia Tesla C2070 GPU. In practice, the GPU performance of the scalar advection kernel is worse because of the high cost of moving the data. If we remove this cost – effectively simulating the case of running a fully integrated model on the GPU – we see that the GPU performance is much better than the original (single-threaded) code.

The difference in performance between the GPU and the Intel CPU is a result of a combination of factors: the CPU code is vectorized but single-threaded; the GPU code is parallelized over multiple

DOI: 10.1002/cpe

compute units, but the threads within a single compute units cannot deliver the theoretical level of parallelism because the memory accesses are not entirely coalesced.

We must introduce another key factor in performance comparisons, often overlooked: the influence of the compiler. To evaluate the performance of the original Fortran code on the Intel Xeon system, we used both the GNU compiler and the commercial PGI compiler, and we found that the code compiled with the latter runs much faster than with the former. The reason is that the PGI compiler makes full use of the 256-bit AVX vector instructions, while gcc does not. Unfortunately, we did not have a license for the C/C++ version of the PGI compiler, and as a result, we could not directly evaluate our C++ OpenMP code performance with this compiler. However, it is reasonable to assume that the PGI compiler would produce the same speed-ups for the C++ code as for the Fortran code. As seen from the OpenMP benchmarks (Figure 4), the multi-threaded CPU version could run almost six times faster than the single-threaded version, so compared with that, the GPU would be about two times faster. It is unlikely that the CPU would be actually six times faster: due to the vectorization, effectively, there will be no benefit from hyperthreading, as demonstrated by Saini in [21]. Consequently, the expected figure is closer to four times, so the GPU would be three times faster than the multi-threaded vectorized CPU version.

From Table I it is clear that the CPIs for the Intel and AMD 48-core CPUs and the GPU are very similar; furthermore, Langkamp [22] found very little difference in the WRF MPI performance when compiled using gfortran and the PGI compiler on an AMD Opteron 2384 system. So, using the 48-core AMD system as our reference, we can compare the compute performance of our kernel with that of the original kernel parallelized using MPI. The results are summarized in Figure 13. We see that the compute performance of the OpenCL kernel on the GPU is about three times faster than the original kernel parallelized using MPI on the 48-core AMD CPU.

Thus, we can conclude that the performance of the OpenCL kernel on the GPU would be at least three times faster than the parallelized kernel on the CPU, for GPUs and CPUs with comparable CPIs.

Note that if we had only used the GNU compiler (gcc/gfortran) on the Intel platform, we would have reported a 20× speed-up, and without compiler optimizations, this figure would be even higher. This example illustrates the values of the estimates based on the CPIs, as well as illustrating the differences caused by different coding styles and compilers.

A final point concerns the optimization goal: should the code be optimized for speed or for power? For the individual user, the aim is either to reduce the run time of the simulation or increase the accuracy. The limitation for an individual user is usually the cost of purchasing the system, rather than the operating cost. Considering the low cost of a GeForce GPU, it might be more cost-effective to buy a GPU rather than an additional multicore CPU system.

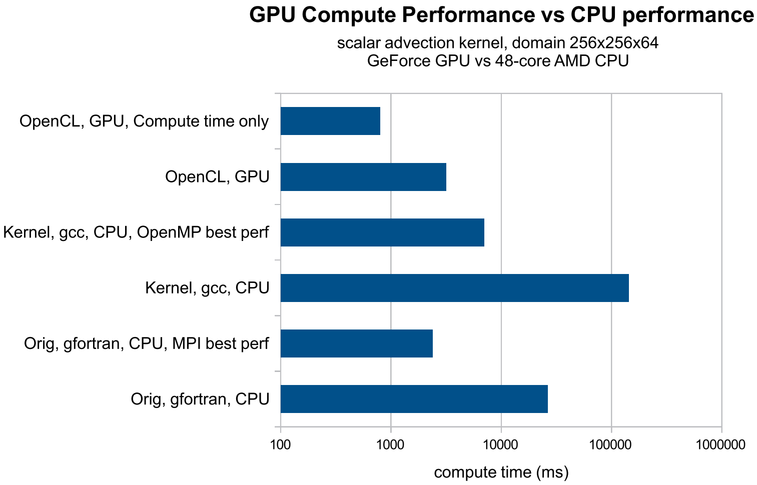

Figure 13. Advection kernel graphics processing unit (GPU) compute performance versus original kernel parallelized using Message Passing Interface (MPI).

DOI: 10.1002/cpe

However, for large high-performance computing centers, the aim is to minimize the energy consumption of the system, because electricity bills are the dominant component in the total cost of ownership. To save energy, one must consider both the power consumption and the speed of execution. For example, if a GPU has the same power consumption as its host CPU, then using it will result in a net energy savings only if the speed-up is greater than a factor of two. Therefore, arguably, the key indicator for assessing hardware acceleration should be the increase in performance-per-watt. Here, the PCIe-hosted GPU is at a disadvantage because it can not work without its host, and even in idle mode, the power consumption of a large multicore CPU is considerable. Hosting the GPU on a low-power ARM or Atom-based system is a possible option to alleviate this issue. A GPU-CPU hybrid such as the AMD Fusion or a very low-power FPGA system could potentially be an even better choice.

## 8. CONCLUSIONS

Based on our work, we can conclude that GPU acceleration of NWP codes such as WRF is both feasible and worthwhile, but that a number of important issues remain to be addressed.

An important conclusion is that rewriting the code as OpenCL-style data-parallel kernels can already result in significant speed-up of the code on a multicore CPU system using either OpenCL or OpenMP, that is, without using a GPU. Consequently, this is an essential step. However, in particular for WRF, this requires a major rewrite of the dynamics and physics code.

Another important finding is that the current PCIe-based CPU–GPU system architecture is sub-optimal for NWP acceleration because of the huge bottleneck of the data transfers over the PCIe bus. On the one hand, this means that a considerable part of the code must be executed on the GPU to amortize this cost. On the other hand, it means that the new CPU–GPU hybrid chips could be very promising for NWP acceleration.

Our final assessment is that even with the current system architectures, accelerating WRF with a factor of up to five times is definitely an achievable goal.

It is important to note that our findings are more generally applicable to multi-physics fluid dynamics codes: in many fluid dynamics codes, the numerical schemes of the advection terms are based on finite differences between neighboring cells, similar to the WRF code. For fluid systems including multi-physics processes, there are many calls to these advection routines. Hence this class of numerical codes will benefit from hardware acceleration.

Accelerating multi-physics fluid dynamics codes including NWP codes is critically important for forecasting applications relating to atmospheric and environmental issues. Forecasting of extreme weather events, early warning of environmental pollution, and emergency response to the dispersion of hazardous materials all require fast and accurate computation of multi-physics atmospheric motion. For example, numerical forecasting of micro-scale atmospheric motion in urban areas and/or over complex topographies should benefit from computational acceleration because it requires a coupling approach merging NWP and computational fluid dynamics codes [23] or very high resolutions to accurately represent a complex topography [24, 25]. Furthermore, computational acceleration would be advantageous in climate prediction simulations with high-resolution global-scale and regional-scale atmospheric models to better represent tropical cyclones and heavy rainfall systems [26, 27]. For all these application, an OpenCL-based parallelization and acceleration approach as presented in this work would be worthwhile.

## REFERENCES

1. World Economic Forum. *Global Risks 2011 Sixth Edition – An initiative of the Risk Response Network*: World Economic Forum, 2011.
2. Stern N. *The Economics of Climate Change: The Stern Review*. Cambridge Univ Pr: London, 2007.
3. Rosenzweig C, Iglesias A, Yang X, Epstein P, Chivian E. Climate change and US agriculture: The impacts of warming and extreme weather events on productivity, plant diseases, and pests. *Center for Health and the Global Environment, Harvard Medical School*, Boston, MA, USA, 2000.
4. Luber G, McGeehin M. Climate change and extreme heat events. *American Journal of Preventive Medicine* 2008; **35**(5):429–435.

5. Knutti R, Furrer R, Tebaldi C, Cermak J, Meehl G. Challenges in combining projections from multiple climate models. *Journal of Climate* 2010; **23**(10):2739–2758.
6. Michalakes J, Chen S, Dudhia J, Hart L, Klemp J, Middlecoff J, Skamarock W. Development of a next generation regional weather research and forecast model. *Developments in Teracomputing: Proceedings of the Ninth ECMWF Workshop on the use of High Performance Computing in Meteorology*, Vol. 1: World Scientific, Reading, UK, 2001; 269–276.
7. Skamarock WC, Klemp JB, Dudhia J, Gill DO, Barker DM, Duda MG, Huang X-Y, Wang W, Powers JG. A description of the Advanced Research WRF Version 3. *Technical Report*, NCAR/TN-475+STR: Boulder, Colorado, US, 2008.
8. Skamarock WC, Klemp JB. A time-split nonhydrostatic atmospheric model for weather research and forecasting applications. *Journal of Computational Physics* 2008; **227**:3465–3485.
9. Michalakes J, Vachharajani M. GPU acceleration of numerical weather prediction. *IEEE International Symposium on Parallel and Distributed Processing, 2008. IPDPS 2008*: IEEE, Miami, Florida, US, 2008; 1–7.
10. Mielikainen J, Huang B, Huang H, Goldberg MD. Improved GPU/CUDA based parallel weather and research forecast (WRF) Single Moment 5-class (WSM5) cloud microphysics. *IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing* 2012; **5**(4):1256–1265.
11. Mielikainen J, Huang B, Huang HL, Goldberg M. GPU acceleration of the updated Goddard shortwave radiation scheme in the weather research and forecasting (WRF) model. *IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing* 2012; **5**(2):555–562.
12. Govett M, Middlecoff J, Henderson T, Rosinski J, Madden P. Successes and challenges porting weather and climate models to GPUs. *AGU Fall Meeting Abstracts*, Vol. 1, San Francisco, California, US, 2011; 02.
13. Fuhrer O, Gysi T, Lapillonne X, Osuna C, Cumming B, Sawyer W, Messme P, Schroeder T, Schulthess TC. GPU consideration for next generation weather and climate simulations. *Technical Report*, CSCS Swiss National Supercomputing Centre: Lugano, Switzerland, 2012.
14. Shimokawabe T, Aoki T, Muroi C, Ishida J, Kawano K, Endo T, Nukada A, Maruyama N, Matsuoka S. An 80-fold speedup, 15.0 TFlops full GPU acceleration of non-hydrostatic weather model ASUCA production code. *2010 International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*: IEEE, New Orleans, Louisiana, US, 2010; 1–11.
15. Munshi A, *et al.*. The OpenCL specification. *Technical Report*, Khronos OpenCL Working Group, 2009.
16. Hong SY, Lim JO. The WRF Single-Moment 6-class microphysics scheme (WSM6). *Journal of the Korean Meteorological Society* 2006; **42**:129–151.
17. Shainer G, Liu T, Michalakes J, Liberman J, Layton J, Celebioglu O, Schultz SA, Mora J, Cownie D. Weather research and forecast (WRF) model performance and profiling analysis on advanced multi-core HPC clusters. *The 10th LCI InternationalConference on High-Performance Clustered Computing*, Boulder, CO, 2009; 1– 14.
18. Morton D, Nudson O, Stephenson C. Benchmarking and evaluation of the weather research and forecasting (WRF) model on the Cray xt5. *Cray user Group Proceedings*, Atlanta, GA, 2009; 04–07.
19. Michalakes J, Dudhia J, Gill D, Henderson T, Klemp J, Skamarock W, Wang W. The weather research and forecast model: software architecture and performance. *Proceedings of the 11th ECMWF Workshop on the use of High Performance Computing in Meteorology*, Vol. 25: World Scientific, Reading, UK, 2004; 1–13.
20. Porter A, Ashworth M, Gadian A, Burton R, Connolly P, Bane M. WRF Code Optimisation for Meso-Scale Process Studies (WOMPS) dCSE Project Report, STFC: Edinburgh, UK, 2012.
21. Saini S, Jin H, Hood R, Barker D, Mehrotra P, Biswas R. The impact of hyper-threading on processor resource utilization in production applications. *2011 18th International Conference on High Performance Computing (HIPC)*: IEEE, Bangalore, India, 2011; 1–10.
22. Langkamp T, Böhner J. Influence of the compiler on multi-CPU performance of WRFv3. *Geoscientific Model Development* 2011; **4**(3):611–623.
23. Nakayama H, Takemi T, Nagai H. Large-eddy simulation of urban boundary-layer flows by generating turbulent inflows from mesoscale meteorological simulations. *Atmospheric Science Letters* 2012; **13**(3):180–186.
24. Takemi T. High-resolution meteorological simulations of local-scale wind fields over complex terrain: a case study for the eastern area of fukushima in march 2011. *Theoretical and Applied Mechanics Japan* 2013; **61**:3–10.
25. Oku Y, Takemi T, Ishikawa H, Kanada S, Nakano M. Representation of extreme weather during a typhoon landfall in regional meteorological simulations: A model intercomparison study for typhoon Songda (2004). *Hydrologic Research Letters* 2010; **4**:1–5.
26. Mizuta R, Yoshimura H, Murakami H, Matsueda M, Endo H, Ose T, Kamiguchi K, Hosaka M, Sugi M, Yukimoto S, Kusunoki S, Kitoh A. Climate simulations using MRI-AGCM3.2 with 20-km grid. *Journal of the Meteorological Society of Japan* 2012; **90A**:233–258.
27. Kanada S, Nakano M, Kato T. Projections of future changes in precipitation and the vertical structure of the frontal zone during the Baiu season in the vicinity of Japan using a 5-km-mesh regional climate model. *Journal of the Meteorological Society of Japan* 2012; **90A**:65–86.