Twinned Buffering: A Simple and Highly Effective Scheme for Parallelization of Successive Over-Relaxation on GPUs and Other Accelerators

Wim Vanderbauwhede
School of Computing Science
University of Glasgow
Glasgow, UK

Tetsuya Takemi
Disaster Prevention Research Institute
~~University of Kyoto~~
Kyoto, Japan

***Index Terms*—General-Purpose computation on Graphics Processing Units (GPGPU), Parallelization of Simulation, Large Scale Scientific Computing**

***Abstract*—In this paper we present a new scheme for parallelization of the Successive Over-Relaxation method for solving the Poisson equation over a 3-D volume. Our new scheme is both simple and effective, outperforming the conventional red-black scheme by a factor of sixteen on an NVIDIA GeForce GTX 590 GPU and by factor of three on an Intel Xeon Phi. We explain the rationale and the implementation in OpenCL and present the performance evaluation results.**

I. Introduction

Numerical Weather Prediction (NWP) models are indispensable tools for weather prediction and the study of weather and climate phenomena. Recently, as a result of climate change, severe weather events have increased both in frequency and severity, and the study and prediction of such events requires higher resolutions to be used in the models, and hence more compute power. As a result, there has been a lot of interest in the use of accelerators such as GPUs to speed up NWP computations [1], [2], [3], [4], [5]. At the heart of every NWP model is a solver for the governing equations. This work concerns an implementation of the Successive Over-Relaxation (SOR) method for solving the Poisson equation in the context of a particular NWP model~~,~~ a Large Eddy Simulator. However, the SOR is a very generic method, and hence the findings in this work are much more widely applicable. In the next section we provide some background on the basic NWP equations and numerical schemes to solve them, and we briefly discuss the Large Eddy Simulator of which our SOR scheme is part. We also discuss the GPU programming technology used, OpenCL, an open standard for heterogeneous computing.

II. Background

A. Use of Successive Over-Relaxation in Numerical Weather Prediction

One of the basic equations used in Numerical Weather Prediction (NWP) is the Navier-Stokes equation given by

$$\begin{aligned} \frac{\partial \mathbf{u}}{\partial t} + \mathbf{u}\nabla\mathbf{u} &= -\frac{1}{\rho}\nabla p + \nu\nabla^2\mathbf{u} - \nabla\mathbf{T} + \mathbf{f} \\ \nabla\mathbf{u} &= 0 \end{aligned} \tag{1}$$

where

p is the pressure
$\mathbf{u}$ is the wind velocity
ρ is the density
ν is the kinematic viscosity
$\mathbf{T}$ the subgrid-scale Reynolds stress
$\mathbf{f}$ the body force (used to model effects of buildings on the flow, see [6])

It is common in NWP codes to solve this equation for the pressure by reducing it first to the Poisson equation, through derivation of both sides:

$$\nabla^2 p = rhs \tag{2}$$

or

$$\left(\frac{\partial^2}{\partial u_x} + \frac{\partial^2}{\partial u_y} + \frac{\partial^2}{\partial u_z}\right) p(x,y,z) = rhs(x,y,z) \tag{3}$$

The Poisson equation can discretised and solved numerically using many different schemes. One of the most popular ones is the Successive Over-Relaxation method, which can be considered as an improvement over the Jacobi method. It has $O(N\sqrt{N})$ serial time and $O(\sqrt{N})$ ideal parallel time (i.e. assuming a PRAM machine with N processors and no communication cost) [some REF to a numerical recipes book]

The term "over-relaxation" refers to the use of the factor $\omega > 1$ which results in faster convergence.

The canonical scheme for implementing the SOR is the red-black scheme, so called because conceptually ~~is~~ is obtained by coloring the points so that any black point only has red direct neighbours and vice versa. In this way, by alternatingly updating the red and black points, the computations can be

Algorithm 1 Successive Over-Relaxation computation of *p*

```
p(i,j,k) = p(i,j,k) + omega*( &
         p(i+1,j,k) + &
         p(i-1,j,k) + &
         p(i,j+1,k) + &
         p(i,j-1,k) + &
         p(i,j,k+1) + &
         p(i,j,k-1) - &
         rhs(i,j,k))/6 &
         - p(i,j,k))
```

Algorithm 2 Red-Black SOR code as used in the Large Eddy Simulator

```
do while (sorr_err > sor_conv )
  sor_err = 0.0
  do nrd = 0,1
    do k = 1,km
      do j = 1,jm
        do i = 1+mod(k+j+nrd,2),im,2
          p_corr = omega*(cn1(i,j,k) *( &
            cn2l(i)*p(i+1,j,k) + &
            cn2s(i)*p(i-1,j,k) + &
            cn3l(j)*p(i,j+1,k) + &
            cn3s(j)*p(i,j-1,k) + &
            cn4l(k)*p(i,j,k+1) + &
            cn4s(k)*p(i,j,k-1) - &
            rhs(i,j,k))-p(i,j,k))
          p(i,j,k) = p(i,j,k) + p_corr
          sor_err = sor_err + p_corr*p_corr
        end do
      end do
    end do
    call boundp1(im,jm,km,p)
  end do
  call boundp2(im,jm,km,p)
  sor_err = sqrt(sor_err)
end do
```

decoupled and parallelised [some REF to a numerical recipes book].

B. The Large Eddy Simulator for Urban Flows

The Large Eddy Simulator for the Study of Urban Boundary-layer Flows (LES) is developed by Hiromasa Nakayama and Haruyasu Nagai at the Japan Atomic Energy Agency and Prof. Tetsuya Takemi at the Disaster Prevention Research Institute of Kyoto University [6], [7]. It generates turbulent flows by using mesoscale meteorological simulations, and was designed to explicitly represent the urban surface geometry (via the τ and *f* terms in the Navier-Stokes equation, cf. [6]). Its purpose is to conduct building-resolving large-eddy simulations (LESs) of boundary-layer flows over urban areas under realistic meteorological conditions. The Weather Research and Forecasting model (WRF, REF) is used to compute the wind profile as input for LES.

In the original LES, the red-black scheme was implemented as follows (the *cn** arrays are coefficients for dealing with a non-uniform grid)

Algorithm 3 Boundary conditions for *i* and *j* used in the Large Eddy Simulator

```
subroutine boundp1(im,jm,km,p)
  integer, intent(In) :: im,jm,km
  real(kind=4), dimension(0:ip+2,0:jp+2,0:kp+1) , in
    do k = 0,km+1
      do j = 0,jm+1
        p(   0,j,k) = p(1 ,j,k)
        p(im+1,j,k) = p(im,j,k)
      end do
    end do
    do k = 0,km+1
      do i = 0,im+1
        p(i,   0,k) = p(i,jm,k)
        p(i,jm+1,k) = p(i, 1,k)
      end do
    end do
  end subroutine boundp1
```

local cach

Algorithm 4 Neumann *k*-boundary used in the Large Eddy Simulator

```
subroutine boundp2(im,jm,km,p)
  integer, intent(In) :: im,jm,km
  real(kind=4), dimension(0:ip+2,0:jp+2,0:kp+1) ,
  do j = 0,jm+1
    do i = 0,im+1
      p(i,j,   0) = p(i,j,1)
      p(i,j,km+1) = p(i,j,km)
    end do
  end do
end subroutine boundp2
```

Conceptually, the *p* array is divided in red and black points so that every red point has black nearest neighbors and vice-versa. The new values for *p* are computed in two iterations (the *nrd*-loop in the code example), one for the red, one for the black.

The calls to boundp1 and boundp2 deal with the boundary conditions. The routine *boundp1* implements a periodic boundary condition in the *j*-direction and an open Neumann condition ($\partial p/\partial x_i = 0$) in the *i*-direction.

The routine *boundp2* implements an open Neumann condition in the *k*-direction.

C. GPU Acceleration of the LES

The LES computation is comparatively very time consuming: for every time step of WRF it performs 120 time steps, and at a much higher spatial resolution. Consequently, in order to benefit from the coupling of WRF and the LES, GPU acceleration is very attractive. It is within this context that our work on the SOR method is positioned. A profiling analysis of the LES shows that the SOR computation dominates the total run time: already for as little as 50 iterations, it acounts for 70% of the total run time.

GPUs have great potential for data-parallel computation but the current generation suffers from being a peripheral

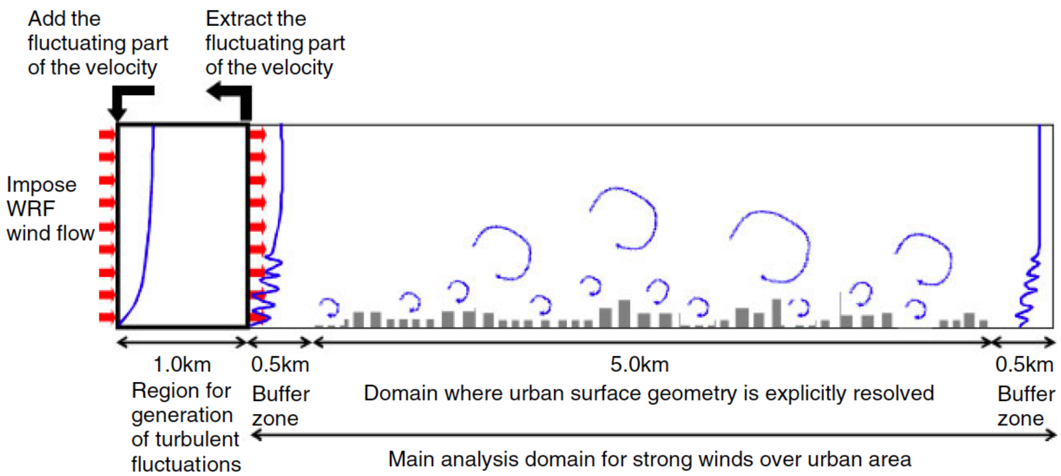

Fig. 1. Coupling WRF and the Large Eddy Simulator

on the PCI Express bus, which has a relatively high latency and much lower bandwidth compared to the main memory of the host computer (see. e.g [REF own work]). For that reason, it is important to limit the host/GPU communication as much as possible. A full discussion of our approach to GPU acceleration of the LES will be published elsewhere, but the overall approach is to keep the velocity and pressure arrays are resident in GPU memory for the full duration of the run, and to control only the transitions between the kernels.

D. Existing GPU Implementations of the SOR algorithm

While the red-black scheme is very effective for single-threaded code, and in fact also for parallel code on distributed memory systems where the communication time is long compared to the compute time, it suffers from poor locality because the accesses to p are strided.

The effect of poor locality is particularly accute for the 3-D case as the computation of the next iteration requires access to all six neighbors of p. If the cache is large enough it is still possible that all neighbours will be cached, but GPUs have relatively small caches (order of 10^4B L1 cache), so in general not all neighbours will be in the cache. As a result, the threads in each compute unit cannot perform coalesced reads or writes.This has been acknowledged by several authors [8], [9] but interestingly most work on SOR on GPU (e.g. [10], [11], [12]) still uses the red-black scheme as-is, likely because for a 2-D SOR the difference in performance is relatively small compared to the gain in performance obtained by implementing the SOR on GPU.

In[9], Konstantinidis and Cotronis explore a GPU implementation of the 2-D SOR method and conclude that their proposed approach of reordering the matrix elements according to their color results in considerable performance improvement. However, their approach is not readily applicable to our problem because one the one hand we have a 3-D array which is much harder to reorder than a 2-D array (i.e. the cost of reordering is higher) and also, we cannot use the reordered array as-is, so we would incur the high reordering cost twice.

In [8], Philip et al. modify the red-black through the use of texture memory for the read-only values and by copying each thread block's portion of the solution to local memory to reduce conflicts on the global memory. They did not however fundamentally change the memory access pattern or ordering.

The overall gain in performance for both these approaches is about a factor of two compared to the unoptimised 2-D red-black scheme.

E. Basic Concepts of OpenCL

To create a GPU-accelerated version of the LES, and hence also for the SOR scheme, we used the OpenCL framework. OpenCL [13] was developed by the Khronos Group in 2008 as an open standard for parallel programming of heterogeneous systems. 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, Windows) and hardware architectures (multicore CPUs, GPUs, FPGAs). The OpenCL API is defined for C and a C++. In practice, the API is quite fine grained and verbose and requires a lot of boiler plate code to be written. Consequently, it is not straightforward to integrate OpenCL in existing codes, especially for non-computing scientists. To facilitate the integration of the OpenCL code into the existing code base, we developed the OclWrapper library [1] 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

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

comes at no cost in terms of features: the OpenCL API is completely accessible.

OpenCL views the accelerator (e.g. the GPU), which it calls the *device*, as consisting of a number of *compute units* which each have a number of *processing elements*, typically the compute unit corresponds to what NVIDIA calls “streaming multiprocessor” or a core on a CPU, and a processing elements is a thread within a compute unit. Each compute unit in the device can access the shared *global* memory and also has its own *local* memory, which is shared between the processing elements within a compute unit. Finally, each processing element has a *private* memory.

The basic parallelisation construct in OpenCL is the *NDRange* (N-Dimensional Range), and index space which expresses the way the data to be operated on is to be partitioned. The NDRanges allows to partition the total amount of work into work groups (typically a compute unit), and into threads per workgroup.

Essentially, the programmer writes a single-threaded kernel which takes an global and local index from the NDRanges. These indices are used to identify the data in global memory to be used in the computations in each thread.

A key point to be noted is that there is no synchronisation construct across compute units, only across processing elements within a compute unit. Consequently, synchronisation across compute units must be handled by the host.

III. Implementation of Parallel SOR in OpenCL

The overall implementation of the SOR in OpenCL is divided between the host and the device as follows: the host runs the iteration loop and computes the SOR error based on partial results from the kernel. The kernel computes the new values for the pressure and the new partial SOR errors, one per compute unit.

A. The Red-Black Scheme

The loop over *nrd* serves two functions: for nrd=0 and nrd=1, the kernel performs the red/black updates; for nrd=2, it updates the boundary values. The global and local ranges are chosen to have thread-parallel computations over j, work-group-parallel computations over k and sequential computations over i, in order to have the best locality of reference. The ranges for updating the boundary are different as the boundary update is a 2-D computation rather than 3-D. The value of nrd is written to the kernel using the *oclWrite1DIntArrayBuffer* command. The kernel is run using *runOcl*, and the values for the SOR error (1 per work group) are read back using *oclRead1DFloatArrayBuffer* and then accumulated. The OpenCL-specific commands are implemented in the OclWrapper API [REF].

Algorithm 5 Host code for red-black SOR

```
do while (sor > pjuge )
    do nrd = 0,2
        if (nrd < 2) then
            oclGlobalRange = kp*jp
            oclLocalRange =  jp
            ngroups = kp
        else
            oclGlobalRange = (ip+2)*(jp+2);
            oclLocalRange = jp+2
            ngroups = ip+2
        end if
        call oclWrite1DIntArrayBuffer(&
          n_ptr_buf,n_ptr_sz, nrd)
        call runOcl(&
          oclGlobalRange,oclLocalRange)
        if (nrd == 1) then
            call oclRead1DFloatArrayBuffer(&
              chunks_sor_buf,chunks_sor_sz,&
              chunks_sor)
            sor = 0.0
            do ii = 1,ngroups
                sor = sor + chunks_sor(ii)
            end do
            sor = sqrt(sor)
        end if
    end do
end do
```

The kernel code is similar in structure to the origibal Fortran code, but the loops over j and k are replaced by thread- and compute-unit-parallel computations. The partial SOR errors are first computed for each thread, then a barrier synchronisation is performed and the partial SOR error for the compute unit is computed and returned to the host in an array. The computations for the correction on p and the boundary conditions are the same as in the Fortran kernel, therefore in the interest of brevity the code for *calc_p_corr*, *calc_boundp1* and *calc_boundp2* is not shown.

As we will see in Section IV, this implementation of the SOR does result in a speed-up of about a factor of two compared to the original host code.

B. The Twinned Buffering Scheme

As the main barrier to performance is the poor locality of reference of the 3-D red-black SOR, we designed a new scheme. Our first step is to replace the red-black approach by a double-buffer approach, i.e. instead of having a single buffer containing “red” and “black” points, we use two buffers, and alternate them at every iteration. Assuming contiguous allocation, the second buffer will be offset from the first buffer by the size of the 3-D domain, which is typically in the order of $10^6 B$ .Consequently, by itself this approach does not lead to better performance, because the locations in one buffer are unlikely to be cached at the same time as the locations in the other buffer. In fact, we can expect to see worse performance.

However, if we create a single buffer consisting of a vector which contains the corresponding points for each buffer, then we get excellent locality of reference. We call this approach *twinned buffering*, and as we will show in the next section, this simple scheme results in excellent performance. The changes to the above host and kernel code are very small. On the host side, we need to declare a 4-D array for the double buffer; on the kernel side, the p array simply changes from __global float* p to __global float2* p. Furthermore, the kernel now uses the first element of the vector to update the second and vice versa. The double-buffering scheme also allows another optimisation: it is not necessary to update the boundary conditions by copying, instead they can be computed. As on the GPU computation is faster than memory access, this is more efficient.

The complete code can be found on GitHub: https://github.com/wimvanderbauwhede/LES.

Algorithm 6 Kernel code for red-black SOR

```
__kernel void press_sor_kernel(
        __global float* p, __global float *rhs,
        const __global float *cn1,
        const __global float *cn2l, const __global float *cn2s,
        const __global float *cn3l, const __global float *cn3s,
        const __global float *cn4l, const __global float *cn4s,
        __global float *chunks_num,
        __global float *rhsav, __global unsigned int *nrd,
        const unsigned int im, const unsigned int jm, const unsigned int km
        ) {
    __local float sor_chunks[NTH];
    unsigned int gr_id = get_group_id(0);
    unsigned int l_id = get_local_id(0);
    if (*nrd<2) {
        float local_sor = 0.0F;
        unsigned int k = gr_id+1; unsigned int j = l_id+1;
        for (unsigned int i=1 + ((k + j + *nrd) % 2);i<=im;i+=2) {
            float p_corr = calc_p_corr(p,...);
            local_sor += p_corr * p_corr;
        } // loop over i
        calc_boundp1(p,...);
        // partial acc of error over threads in CU
        sor_chunks[l_id] = local_sor;
        barrier(CLK_LOCAL_MEM_FENCE);
        float local_sor_acc = 0.0F;
        for(unsigned int s = 0; s < jm; s++) {
            local_sor_acc += sor_chunks[s];
        }
        // return partial errors per CU
        chunks_num[gr_id] = local_sor_acc;
    } else { // nrd==2
        calc_boundp2(p,...);
    } // nrd
}
```

IV. Results and Discussion

We investigated the performance of our new SOR scheme using several OpenCL platforms and different domain sizes. We took care to optimise the compilation of the reference implementation to have a reliable baseline.

What I have right now is REF on CPU/Old kernel and New kernel on GPU for 1 size]

A. Compilers

The compilers used for the comparison were gfortran 4.8.2 for OpenCL code, as well as pgf77 12.5-0 and ifort 12.0.0 for the reference code. We used the following optimizations for auto-vectorization and auto-parallelization:

- `gfortran -Ofast -floop-parallelize-all -ftree-parallelize-loops=24`
- `pgf77 -O3 -fast -Mvect=simd:256`
- `ifort -O3 -parallel`

We established that the run time of the original Fortran code was the same with all compilers (to within a few %).

B. Hardware platforms

The host platform was an Intel Xeon E5-2620 0 @ 2.00GHz, a 6-core CPU with two-way hyperthreading (i.e.12 threads), with AVX vector instruction support, 32GB memory, 15MB cache, Intel OpenCL v1.2. The GPU platform was an NVIDIA GeForce GTX 590 @ 1.20 GHz, 16 compute units, 1.5GB memory, 256KB cache, NVIDIA OpenCL 1.1 (CUDA 6.5.12). Although we are mainly focused on the GPU implementation, we also e used an Intel Xeon Phi 5110P @ 1.05GHz, 59 cores with 4-way hyperthreading, 8GB memory, 30MB cache, Intel OpenCL for MIC v1.2. Table I shows the hardware performance indicators for these systems. In the table, “cores” is what OpenCL reports as “compute units”. On a CPU this is the number of cores times the hyperthreading factor. By “vector size” we we mean SIMD vectors on a CPU or processing elements on a GPU. We can observe that in terms of FLOP performance one could expect the GPU to outperform the CPU and the MIC to outperform both. Furthermore, we observe that the cache on the GPU is much smaller than on the CPU but of the same order as for the Xeon Phi.

In what follows we denote the original Fortran implementation of the red-black SOR as *REF*, and the OpenCL versions deployed on the host CPU, the GPU and the Xeon Phi as *CPU*, *GPU* and *MIC* respectively.

Algorithm 7 Kernel code for SOR with twinned buffering

```
__kernel void press_sor_kernel_twinned_buffer (
    __global float2* p_db,
    ... (same as red/black)
) {
  __local float sor_chunks[NTH];
  unsigned int gr_id = get_group_id(0);
  unsigned int l_id = get_local_id(0);
  float local_sor_acc = 0.0F;
  float local_sor = 0.0F;
  unsigned int k = gr_id;
  unsigned int i = l_id;
  unsigned int k_lhs = k;
  if (k == 0) { k = 1; }
  if (k == km + 1) { k = km; }
  if (i == 0) { i = 1; }
  if (i == im + 1) { i = im; }
  for (unsigned int j_lhs= 0; j_lhs <= jm + 1; j_lhs++) {
    unsigned int j = j_lhs;
    if (j_lhs == 0) { j = jm; }
    if (j_lhs == jm + 1) { j = 1; }
    float p_corr = calc_p_corr_db(p_db,...);
    local_sor += p_corr * p_corr;
  }
  sor_chunks[l_id] = local_sor;
  barrier(CLK_LOCAL_MEM_FENCE);
  local_sor_acc = 0.0F;
  for (unsigned int s = 1; s < jm+1; s++) {
    local_sor_acc += sor_chunks[s];
  }
  chunks_num[gr_id] =  local_sor_acc;
}
```

	#cores	vector size	clock speed (GHz)	FLOPS	Memory BW (GB/s)	Cache
Intel Xeon E5-2620	12	8	2.00	192	42.6	15MB
Nvidia GeForce GTX590	16	32	1.2	614	163.9	256KB
Intel Xeon Phi 5110P	236	16	1.05	3965	320	512KB

TABLE I
HARDWARE PERFORMANCE INDICATORS

C. Red-Black versus Twinned Buffering

Figure 2 shows the performance comparison between both schemes on the three OpenCL platforms. The reference code was compiled with auto-vectorisation and auto-parallelisation optimizations to make as much as possible use of the capabilities of the host platform. The performance of the straight port of the original red-black SOR to OpenCL is reasonable on the CPU: the performance gain is a factor of two. However, on the GPU the performance is slightly worse than the reference and on the MIC it is only about 1.5× better. This illustrates our point about the impact of the poor locality of references. The Twinned Buffering scheme performs somewhat better on the CPU, resulting in a speed-up of 2.5× , but as explained, because of the large cache of the host CPU, we did not expect a big increase in performance. On the GPU however, the performance increase is dramatic: the speed-up is more than 15×. The speed-up of the Twinned Buffering scheme compared to the Red/Black scheme on the MIC is reasonable (3× speed-up) but the overall performance (speed-up compared to the CPU reference) might seem somewhat disappointing considering the hardware capability of the device. However, we will discuss this performance in more detail in the next section.

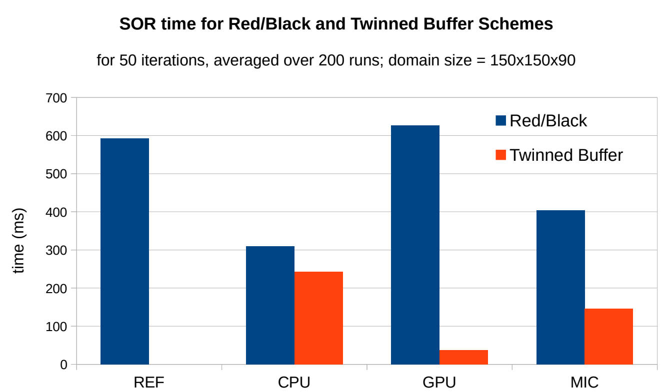

Fig. 2. Comparison of Red/Black and Twinned Buffering schemes on different platforms

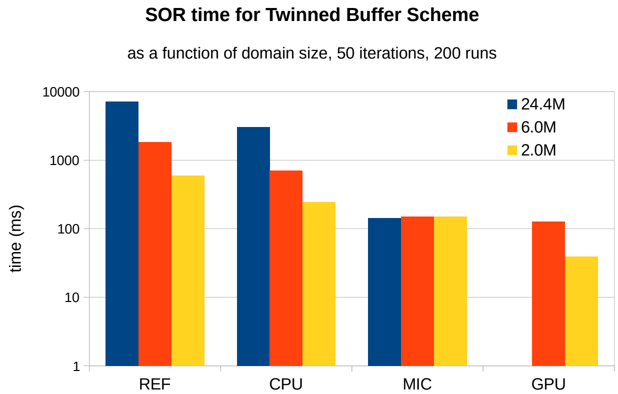

Fig. 3. Performance of Twinned Buffering scheme for different domain sizes

D. Effect of the Domain Size

We evaluated the performance of the Twinned Buffering schemes for different domain sizes, and the results are shown if Fig. 3. For the reference and the OpenCL versions on the CPU and GPU, the performance scales linearly with the domain size. For the GPU, the domain size of 6M points is the maximum that can fit in its global memory (because all arrays required for the LES together take up the complete available memory). The MIC can handle larger domain sizes of up to 24M points, an order of magnitude more than the typical domain size used in the LES simulations. The key observation is that the performance of the MIC is flat over the whole range. The smaller domain sizes under-utilize the MIC's resources, which explains the poor performance observed in the previous section. For the larger domains, the achieved speed-up for the Twinned Buffer scheme is actually 50×, which is much more in line with the hardware capabilities of the device.

V. Conclusions and Future Work

In conclusion, we have presented a novel scheme implementing the 3-D Successive Over-Relaxation (SOR) algorithm for solving the Poisson pressure equation. Though the context of our work is numerical weather prediction, the scheme is much more widely applicable as many problems in science require solving the Poisson equation in three dimensions. The main novelty of our scheme is the use of a buffer of two-element vectors, which we call a *twinned buffer*, to obtain excellent locality if reference. This is particularly important for GPUs, as shown by our results of a speed-up of more than 15×, but the novel scheme leads to improved performance on other OpenCL platforms such as multicore CPUS and the Intel MIC. Thus, our novel scheme offers portable performance over a wide range of OpenCL accelerator platforms. To build on this result we aim to extend the scheme to work across multiple devices.

References

[1] M. Govett, J. Middlecoff, T. Henderson, J. Rosinski, and P. Madden, "Successes and Challenges Porting Weather and Climate Models to GPUs," in *AGU Fall Meeting Abstracts*, vol. 1, 2011, p. 02.

[2] J. Michalakes and M. Vachharajani, "GPU acceleration of numerical weather prediction," in *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*. IEEE, 2008, pp. 1–7.

[3] O. Fuhrer, T. Gysi, X. Lapillonne, C. Osuna, B. Cumming, W. Sawyer, P. Messme, , T. Schroeder, and T. C. Schulthess, "GPU Consideration for Next Generation Weather and Climate Simulations," CSCS Swiss National Supercomputing Centre, Tech. Rep., 2012.

[4] T. Shimokawabe, T. Aoki, C. Muroi, J. Ishida, K. Kawano, T. Endo, A. Nukada, N. Maruyama, and S. Matsuoka, "An 80-fold speedup, 15.0 TFlops full GPU acceleration of non-hydrostatic weather model ASUCA production code," in *High Performance Computing, Networking, Storage and Analysis (SC), 2010 International Conference for*. IEEE, 2010, pp. 1–11.

[5] W. Vanderbauwhede and T. Takemi, "An investigation into the feasibility and benefits of gpu/multicore acceleration of the weather research and forecasting model," in *High Performance Computing and Simulation (HPCS), 2013 International Conference on*. IEEE, 2013, pp. 482–489.

[6] H. Nakayama, T. Takemi, and H. Nagai, "Les analysis of the aerodynamic surface properties for turbulent flows over building arrays with various geometries," *Journal of Applied Meteorology and Climatology*, vol. 50, no. 8, pp. 1692–1712, 2011.

[7] ——, "Large-eddy simulation of urban boundary-layer flows by generating turbulent inflows from mesoscale meteorological simulations," *Atmospheric Science Letters*, vol. 13, no. 3, pp. 180–186, 2012.

[8] S. Philip, B. Summa, V. Pascucci, and P.-T. Bremer, "Hybrid cpu-gpu solver for gradient domain processing of massive images," in *Parallel and Distributed Systems (ICPADS), 2011 IEEE 17th International Conference on*. IEEE, 2011, pp. 244–251.

[9] E. Konstantinidis and Y. Cotronis, "Graphics processing unit acceleration of the red/black sor method," *Concurrency and Computation: Practice and Experience*, vol. 25, no. 8, pp. 1107–1120, 2013.

[10] L. Itu, C. Suciu, F. Moldoveanu, and A. Postelnicu, "Gpu optimized computation of stencil based algorithms," in *Roedunet International Conference (RoEduNet), 2011 10th*. IEEE, 2011, pp. 1–6.

[11] J. T. Liu, Z. S. Ma, S. H. Li, and Y. Zhao, "A gpu accelerated red-black sor algorithm for computational fluid dynamics problems," *Advanced Materials Research*, vol. 320, pp. 335–340, 2011.

[12] C.-W. Hsieh, S.-H. Kuo, F.-A. Kuo, and C.-Y. Chou, "Solving parabolic problems using multithread and gpu," in *Parallel and Distributed Processing with Applications (ISPA), 2010 International Symposium on*. IEEE, 2010, pp. 75–80.

[13] A. Munshi *et al.*, "The opencl specification," *Khronos OpenCL Working Group*, vol. 1, pp. 11–15, 2009.