We would like to thank the anonymous referees' constructive comments. An item-by-item response to the comments is presented below. The manuscript was also revised accordingly with changes/modifications presented below or based on referees' technical suggestions. As a consequence, in our opinion the revised manuscript is improved with respect to the original one.

* ===

Responses to the first anonymous referee's comments:

Anonymous Referee #1
Received and published: 31 January 2015

In this article, the authors port the WRF YSU PBL scheme onto NVIDIA Tesla K40 CPU. With some optimizations, the GPU code gets a good speedup comparing with CPU-only code. I have some concerns about the paper.

(1) First, the NVIDIA Tesla K40 GPUs is the state-of-art accelerator. But the Intel Xeon E5-2603 is not. The comparison may be a little unfair.

The Intel Xeon E5-2603 is the CPU we have here.

(2) Second, the baseline CPU code has not been well tuned. At least the optimization with height dependence release can also be exploited on CPU.

We ran the CPU-based code on one CPU core using exactly the same optimization along with height dependence release as the GPU-based code. It took 1204 ms, which corresponded to speedup of 1.5x.

We added one sentence with a little modification in the third paragraph of Section 5 "Summary and future work" to reflect this issue. The whole paragraph becomes:

"Using one NVIDIA Tesla K40 GPU in the case without I/O transfer, our optimization efforts on the GPU-based YSU PBL scheme can achieve a speedup of 193× with respect to one CPU core, whereas the speedup for one CPU socket (4 cores) with respect to one CPU core is only 3.5×. We also ran the CPU-based code on one CPU core using exactly the same optimization along with height dependence release as the GPU-based code, and its speedup is merely 1.5x as compared to its original Fortran counterpart. In addition, we can even boost the GPU-based speedup to 360× with respect to one CPU core when two K40 GPUs are applied; in this case, one minute of model execution on dual Tesla K40 GPUs will achieve the same outcome as six hours of execution on a single core CPU."

(3) Third, using share memory is a general technique on GPU. According the Section 4.2, you just simply use more L1 cache to get better performance. Have you tried to use share memory for the better locality?

We have tried "cudaFuncCachePreferShared" for using more shared memory as opposed to "cudaFuncCachePreferL1" for using more L1 cache. It was found that the GPU runtime with "cudaFuncCachePreferShared" is almost the same as that with turning off "cudaFuncCachePreferL1" for this scheme.

In other words, the GPU runtimes for using "cudaFuncCachePreferShared" or turning off "cudaFuncCachePreferL1" are liasted in Table 1:

	GPU runtime	Speedup
Non-coalesced	36.0 ms	50.0x
Coalesced	34.2 ms	52.6x

And, the GPU runtimes for using "cudaFuncCachePreferL1" are presented in Table 2:

	GPU runtime	Speedup
Non-coalesced	34.3 ms	52.5x
Coalesced	33.0 ms	54.5x

We added one sentence to the last paragraph in Section 4.2 to address this issue. The whole paragraph becomes:

"Starting with the first CUDA C version of the YSU PBL scheme, the computing performances with L1 cache command was found to be better than that without this command, while the latter performance was noticed to be almost the same as that using "cudaFuncCachePreferShared" command. This suggests that usage of more L1 cache helps to speed up the CUDA C programs for this scheme. The GPU runtime and speedup are summarized in Table 2 after L1 cache command "cudaFuncCachePreferL1" is launched."

*==

Responses to the second anonymous referee's comments:

Anonymous Referee #2
Received and published: 6 May 2015

Summary:
This paper presents the GPU porting and optimization process for a sub-process of the WRF model. The authors describe the mathematical background of the existing model sub-process, then delve into the wide variety of optimizations made to achieve performance from the GPU version of the code. The authors compare the results of each successive optimization to the performance of the original CPU code.

General Comments:
This paper does a good job presenting the various optimization techniques used to achieve a clearly excellent speedup result. The optimizations used are described well, and

will likely be useful in other domains beyond the scope of accelerating the YSU PBL scheme.

Thank you for the positive remarks.

* ==

Questions:

(1) Is there a specific reason that the Yonsei University PBL scheme was chosen for acceleration instead of any other scheme (none of which were named)?

We have implemented several schemes of the Weather Research and Forecasting (*WRF*) model. Yonsei University PBL (YSU PBL) scheme is one of them. Our goal is to have a GPU-based accelerated WRF model.

(2) Is this the most popular model, or perhaps the most amenable to GPU acceleration? I believe some discussion of other schemes may be warranted, at least as motivation for why the YSU scheme was chosen.

The physical feature of no interaction among horizontal grid point (i.e., *i* and *j* in the code) makes all schemes of WRF physical (not dynamical) models very favorable for parallel processing. Among all PBL schemes, YSU PBL is one of the popular schemes used by WRF users, and thus we first chose to work on this scheme among all PBL schemes.

(3) The paper describes a direct mapping from the benchmark dataset's spatial grid size to the implementation's use of thread blocks. Will the accelerated code be easy to apply to other domains or resolutions and other datasets?

Yes. Our code is designed for any kind of data domain and resolution so that the users do not need to adjust the code. The users only need to provide the data dimension, i.e., (i, j, k), where (i, j) is the horizontal grid-point dimension while k is the vertical levels. For example, in our case, $(i, j) = (433, 308)$ and $k = 35$.

(4) Has some other test dataset been used to examine speedups for a different test case?

For this scheme, we have not tried some other test dataset. Nevertheless, it is expected that the speedup should be about the same regardless of which dataset or data size is used.

(5) Is this work intended to be incorporated in some future release of WRF?

Yes.

(6) Is there possibly a timeline for when an accelerated (or partially accelerated) version of WRF is available?

This will depend on the funding and man power though we aim for finishing the implementation of all schemes of the WRF model as soon as possible.

(7) How much does the impressive speedup obtained for the PBL scheme improve the performance of the full WRF model?

Based on those schemes that we have finished the GPU-based implementation, it was found that different scheme has different speedup. Since we have not completed all schemes yet, at this moment it is hard to examine how much this GPU-based YSU PBL scheme would improve the performance of the full WRF model.

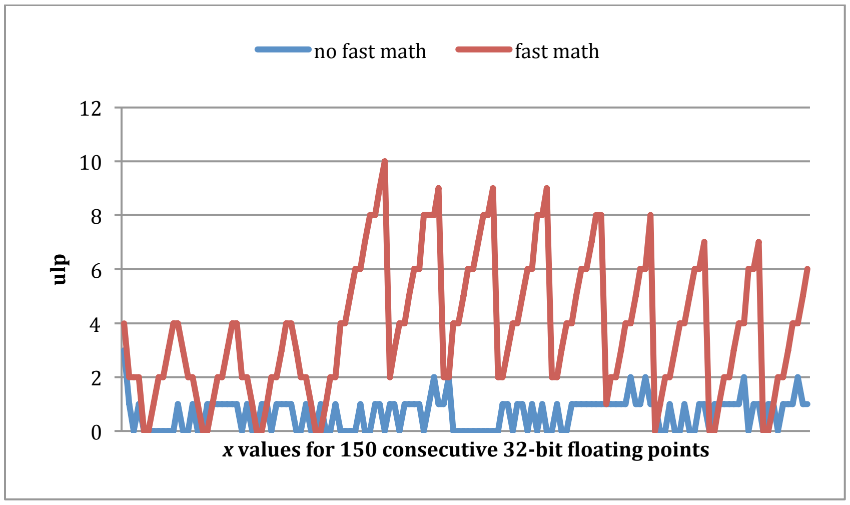
(8) It would be good to see some timing data for the full model with the accelerated PBL scheme incorporated.

Once all schemes of the WRFL model have been implemented on GPUs, we will definitely investigate the performance of the GPU-based WRF model with the accelerated YSU PBL scheme incorporated using timing data.

* ==

Technical Corrections:
Page 8033, line 18: "GPU-accelerated longwave radiation scheme of the rapid radiative transfer model for general circulation models"
Page 8034, line 8: "which is one of the physical models in WRF."
Page 8041, line 12: "built in the WRF model."
Page 8042, line 15: "The driver, in the C language,"
Page 8042, line 20: "into the memory of the CPU."
Page 8042, line 21: "From the viewpoint of CUDA Programming,"
Page 8043, line 10: "contiguous data, and are aligned in memory."
Page 8043, line 24: "Three major reasons for doing this in this way are"
Page 8043, line 26: "CUDA C programs in a short time."
Page 8044, line 3: "accross the entire US. The WRF domain is"
Page 8045, line 6: "one way to do this is to call"
Page 8045, line 20: "to a parallel GPU basis in the next section."
Page 8046, line 6: "one CPU core of an Intel Xeon E5-2603."
Page 8047, line 22: "to the structure of the WRF model,"
Page 8049, line 3: "per thread at 63."
Page 8049, line 15: "make execution more efficient."
Page 8049, line 16: "of the GPU architecture."
Page 8051, line 19: "In the plot, x starts from value"
Page 8062, Table 6a: Column headings need some label relating to what they represent, as opposed to just the description in the caption.
Page 8063, Table 6b: Same as for Table 6a, needs column headings.
Page 8076, Figure 12: Include labels for the x-axis of the graph, instead of just the range mentioned in the paper.

Thank you for the technical corrections. We have made these corrections marked in "red" color in the revised manuscript that would be submitted to the GMD journal. In addition, Fig. 12 has been also modified with adding the x-axis labels for the graph (see below).



Manuscript prepared for Geosci. Model Dev. Discuss.
with version 2015/04/24 7.83 Copernicus papers of the LaTeX class copernicus.cls.
Date: 8 June 2015

Development of efficient GPU parallelization of WRF Yonsei University planetary boundary layer scheme

M. Huang[1], J. Mielikainen[1], B. Huang[1], H. Chen[1], H.-L. A. Huang[1], and M. D. Goldberg[2]

[1]Space Science and Engineering Center, University of Wisconsin-Madison, Madison, USA
[2]NOAA/JPSS, Lanham, MD 20706, USA

Correspondence to: B. Huang (bormin.huang@ssec.wisc.edu)

Abstract

The planetary boundary layer (PBL) is the lowest part of the atmosphere and where its character is directly affected by its contact with the underlying planetary surface. The PBL is responsible for vertical sub-grid-scale fluxes due to eddy transport in the whole atmospheric column. It determines the flux profiles within the well-mixed boundary layer and the more stable layer above. It thus provides an evolutionary model of atmospheric temperature, moisture (including clouds), and horizontal momentum in the entire atmospheric column. For such purposes, several PBL models have been proposed and employed in the weather research and forecasting (WRF) model of which the Yonsei University (YSU) scheme is one. To expedite weather research and prediction, we have put tremendous effort into developing an accelerated implementation of the entire WRF model using Graphics Processing Unit (GPU) massive parallel computing architecture whilst maintaining its accuracy as compared to its CPU-based implementation. This paper presents our efficient GPU-based design on WRF YSU PBL scheme. Using one NVIDIA Tesla K40 GPU, the GPU-based YSU PBL scheme achieves a speedup of $193\times$ with respect to its Central Processing Unit (CPU) counterpart running on one CPU core, whereas the speedup for one CPU socket (4 cores) with respect to one CPU core is only $3.5\times$. We can even boost the speedup to $360\times$ with respect to one CPU core as two K40 GPUs are applied.

1 Introduction

The science of meteorology explains observable weather events, and its application is weather forecasting. Nowadays, sophisticated instruments are used for observations from upper air atmosphere. The collected quantitative data about the current state of the atmosphere are then used to predict future states. This requires the aid of equations of fluid dynamics and thermodynamics that are based on laws of physics, chemistry, and fluid motion. The weather research and forecasting (WRF) (http://www.wrf-model.org/index.php) model meets such a requirement, which is a simulation program consisting of several different physical processes and dynamic solvers. It is designed to serve the needs of both operational forecasting and atmospheric research.

The WRF system supports some of the best possible models for weather forecasting. Nevertheless, a continual challenge for timely weather forecasting is forecast model execution speed. This is a challenge even with the fastest supercomputers, in particular for severe weather events.

With the advent of GPU architectures, the execution speed of weather forecasting models can be greatly increased by taking advantage of the parallelism feature of GPUs. GPUs have hundreds of parallel processor cores for execution on tens of thousands of parallel threads. Furthermore, GPUs possess several merits, such as low cost, low power, large bandwidth, and high performance. These make GPUs more effective than a massively parallel system built from commodity CPUs. Usage of GPUs has been applied very successfully to deal with numerous computational problems in various domains, for instance, porting marine ecosystem model spin-up using transport matrices to GPUs (Siewertsen et al., 2013), GPU-accelerated longwave radiation scheme of the rapid radiative transfer model for general circulation models (RRTMG) (Price et al., 2014), advances in multi-GPU smoothed particle hydrodynamics simulations (Rustico et al., 2014), speeding up the computation of WRF double moment 6-class microphysics scheme with GPU (Mielikainen et al., 2013), real-time implementation of the pixel purity index algorithm for endmember identification on GPUs (Wu et al., 2014), fat vs. thin threading approach on GPUs: application to stochastic simulation of chemical reactions (Klingbeil et al., 2012), ASAMgpu V1.0 – a moist fully compressible atmospheric model using graphics processing units (GPUs) (Horn, 2012), GPU acceleration of predictive partitioned vector quantization for ultraspectral sounder data compression (Wei, 2011), clusters vs. GPUs for parallel automatic target detection in remotely sensed hyperspectral images (Paz et al., 2010), a GPU-accelerated wavelet decompression system with SPIHT and Reed-Solomon decoding for satellite images (Song, 2011), to name several.

To expedite weather analysis research and forecasting, we have put tremendous efforts into developing an accelerated implementation of the entire WRF model using Graphics

Processing Unit (GPU) massive parallel architecture whilst maintaining its accuracy as compared to its CPU-based implementation. This study develops an efficient GPU-based design on the Yonsei University (YSU) planetary boundary layer scheme, which is one of physical models in WRF. The PBL is responsible for vertical sub-grid-scale fluxes due to eddy transports in the whole atmospheric column. It determines the flux profiles within the well-mixed boundary layer and the more stable layer above, and thus provides an evolutionary model of atmospheric temperature, moisture (including clouds), and horizontal momentum in the entire atmospheric column.

This paper is structured as follows. Section 2 describes YSU PBL scheme. Section 3 outlines the GPU hardware specification as well as a brief description of the basis of CUDA computing engine for GPUs. The GPU-based implementation is also given in Sect. 3. The development of optimizing the GPU-based YSU PBL scheme is presented in Sect. 4. Summary and future work is given in Sect. 5.

2 YSU PBL scheme

The scheme of YSU is one of the PBL models in WRF. The PBL process is illustrated in Fig. 1. Based on reference (Hong et al., 2006), a brief description of the YSU PBL scheme is presented below.

2.1 Mixed-layer diffusion

The momentum diffusivity coefficient is formulated based on the work done in references (Troen et al., 1986; Hong et al., 1996; Noh et al., 2003):

$$K_m = k w_s z \left(1 - \frac{z}{h}\right)^{\mathsf{p}}, \tag{1}$$

where p is the profile shape exponent taken to be 2, k the von Karman constant ($= 0.4$), z the height from the surface, and h the PBL height. The mixed-layer velocity scale is given as

$$w_s = \left(u_*^3 + \varphi_m k w_{*b}^3 z/h\right)^{1/3}, \tag{2}$$

where u_* is the surface frictional velocity scale, φ_m the wind profile function evaluated at the top of the surface layer, and w_{*b} the convective velocity scale for the moist air which is defined as

$$w_{*b} = \left[(g/\theta_{va})\left(w'\theta_v'\right)_0 h\right]^{1/3}.$$

The counter-gradient term for θ and momentum is given by

$$\lambda_c = b\frac{\left(\overline{w'\theta_v'}\right)_0}{w_{s0}h}, \tag{3}$$

where $(\overline{w'\theta_v'})_0$ is the corresponding surface flux for θ, u, and v, and b the coefficient of proportionality which will be derived below. The mixed-layer velocity scale w_{s0} is defined as the velocity at $z = 0.5h$ in Eq. (2).

The eddy diffusivity for temperature and moisture K_t is computed from K_m in Eq. (1) by using the relationship of the Prandtl number (Noh et al., 2003), which is given by

$$Pr = 1 + (Pr_0 - 1)\exp[-3(z - \varepsilon h)^2/h^2], \tag{4}$$

where $Pr_0 = (\varphi_t/\varphi_m + bk\varepsilon)$ is the Prandtl number at the top of the surface layer given by references (Troen et al.,1986; Hong et al., 1996). The ratio of the surface layer height to the PBL height, ε, is specified to be 0.1.

To satisfy the compatibility between the surface layer top and the bottom of the PBL, identical profile functions are used to those in surface layer physics. First, for unstable and neutral conditions $[(\overline{w'\theta'_v})_0 \geq 0]$,

$$\varphi_m = \left(1 - 16\frac{0.1\,\mathrm{h}}{L}\right)^{-1/4} \quad \text{for } u \text{ and } v, \tag{5a}$$

$$\varphi_t = \left(1 - 16\frac{0.1h}{L}\right)^{-1/2} \quad \text{for } \theta \text{ and } q, \tag{5b}$$

while for the stable regime $[(\overline{w'\theta'_v})_0 < 0]$,

$$\varphi_m = \varphi_t = \left[1 + 5\frac{0.1h}{L}\right], \tag{6}$$

where h is, again, the PBL height, and L the Monin Obukhov length scale. To determine the factor b in Eq. (3), the exponent of $-1/3$ is chosen to ensure the free convection limit. Typically L ranges from -50 to 0 in unstable situations. Therefore, we can use the following approximation:

$$\varphi_m = \left(1 - 16\frac{0.1h}{L}\right)^{-1/4} \simeq \left(1 - 8\frac{0.1h}{L}\right)^{-1/3}. \tag{7}$$

Following the work in references (Noh et al., 2003; Moeng et al., 1994), the heat flux amount at the inversion layer is expressed by

$$\left(\overline{w'\theta'}\right)_h = -e_1 w_m^3/h, \tag{8}$$

where e_1 is the dimensional coefficient (= 4.5 $\mathrm{m}^{-1}\,\mathrm{s}^2\mathrm{K}$), w_m the velocity scale based on the surface layer turbulence ($w_m^3 = w_*^3 + 5u_*^3$), and the mixed-layer velocity scale for the dry air $w_* = [(g/\theta_a)(\overline{w'\theta'_0})h]^{1/3}$. Using a typical value of θ_a at 300 K, the gravity at 10 ms^{-2}, and the limit of $u_* = 0$ in the free convection limit, Eq. (8) can be generalized for the moist air with a non-dimensional constant, which can be expressed by

$$\left(\overline{w'\theta_v'}\right)_h = -0.15\left(\frac{\theta_{va}}{g}\right) w_m^3/h, \tag{9}$$

where w_m considers the water vapor driven virtual effect for buoyancy flux. Given the buoyancy flux at the inversion layer, Eq. (9), the flux at the inversion layer for scalars θ and q, and vector quantities u and v, are proportional to the change in each variable at the inversion layer:

$$\left(\overline{w'\theta'}\right)_h = w_\mathrm{e}\Delta\theta|_h, \tag{10a}$$

$$\left(\overline{w'q'}\right)_h = w_\mathrm{e}\Delta q|_h, \tag{10b}$$

$$\left(\overline{w'u'}\right)_h = Pr_h w_\mathrm{e}\Delta u|_h, \tag{10c}$$

$$\left(\overline{w'v'}\right)_h = Pr_h w_\mathrm{e}\Delta v|_h, \tag{10d}$$

respectively. Here w_e is the entrainment rate at the inversion layer, which is expressed by

$$w_\mathrm{e} = \frac{\left(\overline{w'\theta_v'}\right)_h}{\Delta\theta_v|_h}, \tag{11}$$

where the maximum magnitude of w_e is limited to w_m to prevent excessively strong entrainment in the presence of too small of a jump in θ_v in the denominator. The Prandtl number at the inversion layer Pr_h is set as 1. Meanwhile, the flux for the liquid water substance at the inversion layer is assumed to be zero.

Following the reference (Hong et al., 1996), h is determined by checking the stability between the lowest model level and levels above considering the temperature perturbation due to surface buoyancy flux, which is expressed by,

$$\theta_v(h) = \theta_{va} + \theta_T\left[= a\frac{(\overline{w'\theta_v'})_0}{w_{s0}}\right], \tag{12}$$

where a is set to 6.8, the same as the b factor in Eq. (3). In Eq. (2), θ_T ranges less than 1 K under clear-sky condition where $\theta_v(h)$ is the virtual potential temperature at h. The quantity

a is an important parameter in the new scheme. Numerically, h is obtained by two steps. First, h is estimated by

$$h = \mathrm{Rib}_{\mathrm{cr}} \frac{\theta_{va}|U(h)|^2}{g[\theta_v(h) - \theta_{\mathrm{s}}]}$$

without considering θ_T, where $\mathrm{Rib}_{\mathrm{cr}}$ is the critical bulk Richardson number, $U(h)$ the horizontal wind speed at h, θ_{va} the virtual potential temperature at the lowest model level, $\theta_v(h)$ the virtual potential temperature at $\theta_v(h)$, and θ_{s} the near the surface temperature. This estimated h is utilized to compute the profile functions in Eqs. (5)–(7), and to compute the w_{s0}, which is estimated to be the value at $z = h/2$ in Eq. (2). Secondly, using w_{s0} and θ_T in Eq. (12), h is revised by checking the bulk stability, Eq. (12), between the surface layer (lowest model level) and levels above. With the revised h and w_{s0}, K_{m} is obtained by Eq. (1), entrainment terms in Eqs. (9)–(11), and K_{t} by the Prandtl number in Eq. (4). The counter gradient correction terms for θ in Eq. (4) are also obtained by Eq. (3).

2.2 Free atmosphere diffusion

The local diffusion scheme, the so-called local K approach (Louis, 1979) is utilized for free atmospheric diffusion above the mixed layer ($z > h$). In this scheme, the effects of local instabilities in the environmental profile and the penetration of entrainment flux above h irrespective of local stability are both taken into account within the entrainment zone and the local K approach above.

In the reference (Noh, 2003), the diffusion coefficients for mass $(t:\theta,q)$ and momentum $(m:u,v)$ are expressed by

$$K_{t\_\mathrm{ent}} = \frac{-\left(\overline{w'\theta_v'}\right)_h}{\left(\partial\theta_v/\partial z\right)_h} \exp\left[-\frac{(z-h)^2}{\delta^2}\right], \tag{13a}$$

$$K_{m\_\mathrm{ent}} = Pr_h \frac{-\left(\overline{w'\theta_v'}\right)_h}{\left(\partial\theta_v/\partial z\right)_h} \exp\left[-\frac{(z-h)^2}{\delta^2}\right], \tag{13b}$$

and the thickness of the entrainment zone can be estimated as

$$\delta/h = d_1 + d_2 \mathrm{Ri}_{\mathrm{con}}^{-1}, \tag{14}$$

where w_m is the velocity scale for the entrainment, $\mathrm{Ri}_{\mathrm{con}}$ the convective Richardson number at the inversion layer: $\mathrm{Ri}_{\mathrm{con}} = [(g/\theta_{va})\,h\,\Delta\theta_{v\_\mathrm{ent}}]/w_m^2$, and constants d_1 and d_2 are set as 0.02 and 0.05, respectively.

Following references (Noh, 2003; Louis, 1979), the vertical diffusivity coefficients for momentum $(m:u,v)$ and mass $(t;\theta,q)$ above h are represented as,

$$K_{m\_\mathrm{loc},t\_\mathrm{loc}} = l^2 f_{m,t}(\mathrm{Rig})\left(\frac{\partial U}{\partial z}\right), \tag{15}$$

in terms of the mixing length l, the stability function $f_{m,t}(\mathrm{Rig})$, and the vertical wind shear, $|\partial U/\partial z|$. The stability functions $f_{m,t}$ are represented in terms of the local gradient Richardson number Rig. For the non-cloudy layer,

$$\text{Rig} = \frac{g}{\theta_v}\left[\frac{\partial\theta_v/\partial z}{(\partial U/\partial z)^2}\right]. \tag{16a}$$

For the cloudy air, Rig is modified for reduced stability within cloudy air, which is expressed by (Durran et al., 1982)

$$\text{Rig}_c = \left(1+\frac{L_v q_v}{R_d T}\right)\left[\text{Rig} - \frac{g^2}{|\partial U/\partial z|^2}\frac{1}{c_p T}\frac{(A-B)}{(1+A)}\right], \tag{16b}$$

where $A = L_v^2 q_v / c_p R_v T^2$ and $B = L_v q_v / R_d T$. The computed Rig is truncated to -100 to prevent unrealistically unstable regimes. The mixing length scale l is given by

$$\frac{1}{l} = \frac{1}{k_z} + \frac{1}{\lambda_0}, \tag{17}$$

where k is the von Karman constant ($= 0.4$), z the height from the surface, and λ_0 is the asymptotic length scale ($= 150$ m) (Kim et al., 1992). The stability function, $f_{m,t}$(Rig), differ for stable and unstable regimes. The stability formulas from NCEP MRF model (Betts et al.,

1996) are used. For the stably stratified free atmosphere (Rig > 0),

$$f_{m,t}(\mathrm{Rig}) = \frac{1}{(1+5\mathrm{Rig})^2}, \tag{18}$$

and the Prandtl number is given by,

$$Pr = 1.0 + 2.1 = \mathrm{Rig}. \tag{19}$$

For the neutral and unstably stratified atmosphere (Rig ≤ 0),

$$f_t(\mathrm{Rig}) = 1 - \frac{8\mathrm{Rig}}{1+1.286\sqrt{-\mathrm{Rig}}}, \tag{20a}$$

$$f_m(\mathrm{Rig}) = 1 - \frac{8\mathrm{Rig}}{1+1.746\sqrt{-\mathrm{Rig}}}. \tag{20b}$$

For the entrainment zone above h, the diffusivity is determined by geometrically averaging the two different diffusivity coefficients from Eqs. (13) and (14), and is expressed by,

$$K_{m,t} = (K_{m,t\_\mathrm{ent}} K_{m,t\_\mathrm{loc}})^{1/2}. \tag{21}$$

Equation (21) represents not only the entrainment, but also the free atmospheric mixing when the entrainment above the bottom of the inversion layer is induced by vertical wind shear at PBL top. With the diffusion coefficients and counter-gradient correction terms computed in Eqs. (1)–(21), the diffusion equations for all prognostic variables $(C, u, v, \theta, q_v, q_c, q_i)$ expressed as, for example for C,

$$\frac{\partial C}{\partial t} = \frac{\partial}{\partial z}\left[k_c\left(\frac{\partial C}{\partial z} - \gamma_c\right) - \left(\overline{w'c'}\right)_h \left(\frac{z}{h}\right)^3\right] \tag{22}$$

and can be solved by an implicit numerical method (Bright et al., 2002) that has been built in the WRF model. Here C is heat capacity, u and v are horizontal velocity components, θ

is the potential temperature, q_v, q_c, and q_i are the mixing ratios of water vapor, cloud, and cloud ice respectively. Note that the term $-(\overline{w'c'})_h(\frac{z}{h})^3$ is an asymptotic entrainment flux term at the inversion layer and is not included in the MRF PBL model (Hong et al., 1996).

Since there are no interactions among horizontal grid points, the WRF YSU PBL scheme is highly suited to massively parallel processing and great speed advantage can be expected. What follows is a presentation of our GPU-based development on YSU PBL scheme.

3 GPU hardware specification

3.1 GPU device specification and basis of CUDA engine

We developed the massively parallel GPU version of the YSU PBL scheme using NVIDIA Tesla K40 GPUs (NVIDIA Tesla GPU; NVIDIA Tesla K40), while Intel Xeon CPU E5-2603 at 1.8 GHz is used for executing its counterpart CPU-based program for speedup comparison. The clock frequencies of memory and GPU processor (in boost mode) are at 3004 and 875 MHz for K40, respectively. One Tesla K40 GPU has 15 Streaming Multiprocessors (SMX). Each SMX units has 192 CUDA cores, amounting to 2880 cores. The hardware specification employed in our study is depicted in Fig. 2. In contrast, our CPU system has two sockets, each of which has four cores.

CUDA C is an extension to the C programming language and which offers a direct programming in the GPUs. It is designed such that its construction allows for execution in a manner of data-level parallelism. A CUDA program is arranged into two parts: a serial program running on the CPU and a parallel part running on the GPU, where the parallel part is called a kernel. The driver, in the C language, distributes a large number of copies of the kernel into available multiprocessors and executes them simultaneously. The CUDA parallel program automatically utilizes more parallelism on GPUs when there are more processor cores available. A CUDA program consists of three computational phases: transmission of data into the global memory of the GPU, execution of the CUDA kernel, and delivery of results from the GPU into the memory of the CPU.

From the viewpoint of CUDA programming, a thread is the basic atomic unit of parallelism. Threads are organized into a three-level hierarchy. The highest level is a grid, which consists of thread blocks. A grid is a three-dimensional array of thread blocks. A domain of 2-dimensional horizontal grid points, 433×308, is adopted in the YSU PBL scheme (see Sect. 3.2), which implies that 433×308 threads are required if one GPU is used. Each thread executes the whole numerical calculation of equations described in Sect. 2. Given the block size (i.e., number of threads per block) of 64 available, this suggests that one needs 7×308 blocks. Figure 3 illustrates the three-level thread hierarchy of a device for one GPU that was implemented in our study. Thread blocks implement coarse-grained scalable data parallelism and they are executed independently, which permits them to be scheduled in any order across any number of cores. This allows the CUDA program to scale with the number of processors.

The execution of CUDA programs can be achieved more efficiently in GPUs if the global memory is maximized and the number of data transactions is minimized. This can be accomplished through the global memory access by every 16 threads grouped together and coalesced into one or two memory transactions. This is the so-called coalesced memory access, which can be more effective if adjacent threads load or store contiguous data, and are aligned in memory. To enable the coalesced global memory access, the function in NVIDIA CUDA library *"cudaMallocPitch"* is used to pad the data, which would enhance the data transfer sizes between host and device.

Threads of 32 are arranged together in execution, which are called a warp, while global memory loads and stores by half of a warp (i.e., 16 threads). A CUDA program group inside a multiprocessor issues the same instruction to all the threads in a warp. Different global memory accesses are coalesced by the device in as few as one memory transaction when the starting address of the memory access is aligned and the threads access the data sequentially. An efficient use of the global memory is one of the essences for a high performance CUDA kernel.

3.2 GPU-based implementation

The current WRF programs are written in the Fortran language. To develop a GPU-based and parallel implementation, the Fortran programs of YSU PBL scheme are first translated to standard C programs, followed by converting the C into CUDA C that can run on GPUs efficiently. Three major reasons for doing this in this way are (i) to ensure correct results, (ii) to make the difference between C and CUDA C implementations to be very small, and (iii) to allow conversion from C programs to CUDA C programs in a short time.

To test whether the programming of the YSU PBL scheme is correct, we used a CONtinental United States (CONUS) benchmark data set, a 12 km resolution domain, collected on 24 October 2001 (CONUS website). This is a 48 h forecast over the continental US capturing the development of a strong baroclinic cyclone and a frontal boundary that extends from north to south across the entire US. The WRF domain is a geographic region of interest discretized into a 2-dimensional grid point parallel to the ground, which are labeled as (i,j) in the WRF codes and in the following discussion. Each grid point deals with multiple levels, corresponding to various vertical heights in the atmosphere, which is denoted as k in both the programs and in the discussion presented below. The size of the CONUS 12 km domain is 433×308 horizontal grid points with 35 vertical levels. Figure 4 exemplifies the mapping of CONUS domains onto one GPU thread-block-grid domain.

In generating correct C programs from Fortran, followed by the conversion from C to CUDA C, considerable care must be taken. First of all, Fortran array-indexing uses non-zero indices as the first index and these have to be converted into C/CUDA C arrays using zero-based indexing. That is, the first index value of the Fortran arrays in WRF codes is 1 or non-zero value, while that of a C/CUDA C array is 0. Secondly, some temporary array variables are replaced by scalar variables re-computed on GPU memory as needed and stored in local register memory; we call these scalarable. This is done because re-calculating values in GPU threads (i.e., local register memory) is faster than transferring them from a relatively slower global memory.

The third area for special care relates to the handling of the horizontal grid point (i, j) inside the kernel. When translating C programs to CUDA C programs, the loops for spatial grid points (i, j) are replaced by index computations using thread and block indices:

$$i = threadIdx.x + blockIdx.x \times blockDim.x$$

$$j = blockIdx.y$$

where *threadIdx* and *blockIdx* are thread index and block index respectively, and *blockDim* the dimensional size of a block. Each grid point (i, j) represents a thread in CUDA C programs. Hence, there are two purposes for decomposing the domain in this way. One is to make the execution in each thread independent from one another, and the other is to compute values for all vertical levels (i.e., all k components in the CUDA C programs) in one spatial grid position, (i, j). There are no interactions among horizontal grid points, according to the physical model described in Sect. 2, which means that this computationally efficient thread and block layout is permissible.

Fourthly, to check whether a kernel is launched successfully, one way to do this is to call *cudaGetLastError*(). If the kernel launch fails, this command would report error messages right after CUDA kernel launch. Finally, once a kernel is successfully launched, to obtain the correct GPU execution runtime, command *cudaThreadSynchronize*() must be called.

The default compiler options from WRF were used to compile the Fortran and C versions of the YSU PBL scheme. For Fortran programs, we used gfortran along with compiler options: *-O3 -ftree-vectorize -ftree-loop-linear -funroll-loops*. For C programs, we used gcc with compiler options: *-O3 -ftree -vectorize -ftree-loop-linear -funroll-loops -lm*. We have first verified that the outputs of C programs were identical to those of the Fortran programs using the same level of compiler optimization.

When the first CUDA C version of the YSU PBL scheme, directly translated from C version without any optimization, was ready, we examined features of the original Fortran programs to discover further optimization opportunities. We will present the evolution of this scheme from a CPU basis to a parallel GPU basis in the next section.

4 Development of GPU-based YSU PBL scheme

4.1 Premise for optimizing GPU-based YSU PBL scheme

To perform the GPU-based YSU PBL scheme, the CUDA C programs were compiled using nvcc (NVIDIA CUDA compiler) version 6.0 and executed on one Tesla K40 GPU with compute capability 3.5. The compiler options are *–O3 –gpu-architecture sm 35 -fmad=false -m64 –maxrregcount 63 –restrict.* The value of *63* means the number of registers per thread, which was randomly picked at present. Besides, the thread block size (i.e., threads per block) was chosen as 64 at this stage. The effects of block size and registers per thread for performing this scheme will be discussed in Sects. 4.5 and 4.6 respectively. Table 1 lists the runtime and speedup of this first-version GPU-based scheme, where the speedup is calculated as GPU runtime as compared to its CPU counterpart running on one CPU core of an Intel Xeon E5-2603. The same definition of speedup calculation is used in subsequent discussions.

Before we go further to present our optimizations on the GPU-based YSU PBL scheme, a couple of things are worthwhile to mention. The computation of this scheme is merely one intermediate module of the entire WRF module. When WRF is fully implemented on GPUs, the inputs of this intermediate module will not have to be transferred from CPU memory. Instead, they will be results from previous WRF module. Similarly, outputs of this will be inputs to next WRF module. Hence, transfers to/from CPU memory are not involved in this intermediate module during its computation. In the studies presented in Sects. 4.1–4.8, I/O transfer timings are turned off. The results of computing performance with I/O transfer and multi GPUs will be given in Sect. 4.9. In each evolution of optimization, the correctness of CUDA C outputs must be verified in comparison to the original Fortran outputs.

4.2 Optimization with more L1 cache

For each Tesla K40 GPU (see Fig. 2), every SMX has a data cache, which is shared among CUDA cores. The data cache can be configured as 16 KB software-managed cache called shared memory and 48 KB hardware cache (L1) or the other way around, or both can share equally the memory, i.e. 32 KB each.

To employ more L1 cache than shared memory, a command *"cudaFuncCachePreferL1"* was launched in our CUDA C programs. In contrast, without this command indicates less L1 cache than shared memory. Figure 5 depicts the memory allocation between L1 cache and shared memory with and without launching L1 cache command.

Starting with the first CUDA C version of the YSU PBL scheme, the computing performances with L1 cache command was found to be better than that without this command, while the latter performance was noticed to be almost the same as that using *"cudaFuncCachePreferShared"* command. This suggests that usage of more L1 cache helps to speed up the CUDA C programs for this scheme. The GPU runtime and speedup are summarized in Table 2 after L1 cache command *"cudaFuncCachePreferL1"* is launched.

4.3 Optimization with scalarizing temporary arrays

Due to the parallelism architecture in CUDA programs, each thread must have its own local copy of the temporary array data. For instance, for 3-dimensional arrays, when they are converted from CPU-based implementation to GPU-based implementation, they are retyained as 3-dimensional, but 1-dimensional and 2-dimensional arrays that involve with k (height) components must be re-arranged as 3-dimensional arrays in GPU-based implementation; otherwise, the access to the contents of the arrays could be some other computed values from other threads. One special case is that the 2-dimensional arrays with (i,j) elements, are still kept as arrays with (i,j) elements because what each thread deals with is the grid point (i,j). By so arranging arrays, which is necessary in GPU-based implementation, memory usage is considerably increased, and the computing performance degraded. Figure 6 illustrates the concepts of such array re-arrangement.

One solution of such a problem is to reduce the use of loop operation in the CUDA programs by merging several loop operations into one single loop operation. With this approach, some scalarable temporary arrays are replaced by scalar variables. Owing to the structure of the WRF model, this approach can only be applicable to those scalarable temporary arrays in the vertical-level (i.e., k) components, not in the grid-point component (i.e., i or j). The scalar variables are re-computed in fast local memory, rather than in the slower global memory, in order to reduce the access time. This approach of diminishing the number of loop operations is applicable to CPU-based programs. However, the computation are still executed in one single-threaded core by looping through all horizontal grid points (i,j), and it would not much speed up the computing performance. In contrast, this loop-reduction approach is extremely useful for GPU-based programs due to its multi-threaded nature.

Figure 7 illustrates the concept presented here. This reduces the transferring of data from global memory and delivers significant time savings. For this YSU PBL scheme, such replacement of temporary arrays by scalar variables drops the temporary arrays from 68 down to only 14 arrays, and apparently enhances the computing performance by a lot. Table 3 summarizes the GPU runtime and speedup after scalarizing most of temporary arrays.

4.4 Optimization with height dependence release

The remaining 14 local arrays present difficulties because the k components of the vertical heights are not independent with one another. This can be seen from Eq. (22) when one tries to solve the diffusion equations for those prognostic variables $(C,u,v,\theta,q_v,q_c,q_i)$, where there is dependence in the vertical z level (i.e., k component in the codes). These 14 local arrays are involved in the final solution calculation for those prognostic variables. For the first part of these 14 array variables involved, the calculation of the kth component depends on the input of the $(k-1)$th component. This is to calculate the contents inside the brackets in Eq. (22), which is a differential equation as a function of the vertical height and is related to calculations in Eqs. (1)–(21). For the second part, the calculation of the kth component needs inputs from the $(k+1)$th component. This is to carry out the final results for those prognostic variables, which is again a derivative equation with respect to the vertical height.

About one third of the original Fortran programs appears to involve dependencies among $(k-1)$th, kth, and $(k+1)$th components. Figure 8 describes the conceptual idea for how to release the height dependence in order to reduce the access time to the global memory. This is the most time-consuming part in the physical programs. Table 4 gives the GPU runtime and speedup after the release of the height dependence.

4.5 Impact of block size on GPU-based YSU PBL scheme

The impact of the thread block size on the computing performance of the GPU-based YSU PBL scheme was evaluated by varying the block size while keeping the registers per thread at 63. Figure 9 shows the speedup of the optimized CUDA C programs (see Sects. 4.2–4.4) vs. the block size for cases with and without coalesced memory access. To obtain this plot, 16 executions of the CUDA C programs are executed in a row for each given block size. Excluding the first three runtimes due to unstable computing performance, the remaining runtimes are averaged and used for the average speedup calculation for the given block size. From this study, it was found that the block size of 64 could produce the best performance; this is what we used in Sects. 4.1–4.4.

In addition, when the block size is a multiple of 32 threads, i.e., a warp, it was found that the computing performance is better than the neighboring block sizes, which forms a cycle of oscillation every 32 threads. This is understandable because threads of 32 are grouped together and the multiprocessor issues the same instruction to all the threads in a warp in order to make execution more efficient. This is one of the merits of the GPU architecture.

4.6 Impact of registers per thread on GPU-based YSU PBL scheme

By keeping the block size at 64, the optimized CUDA C version of the YSU PBL scheme was studied for speedup vs. the number of registers per thread. Similar to the approach presented in the previous section, given a number of registers per thread, 16 executions of the CUDA C programs are performed and only the last 13 runtimes are used in calculating the average speedup. The results are displayed in Fig. 10 for cases with and without coalesced memory access. This figure shows that the optimal computing performance occurs at 63 registers per thread for this scheme, and the speedups seem to keep dropping beyond this number of registers.

4.7 Data transfer between CPU and GPUs

Usually the time consumed by kernel execution is less than the time occupied by the data transfer between CPU (i.e., host) and GPUs (i.e., devices). Using the optimized CUDA C programs of the YSU PBL scheme along with the optimal registers/thread $= 63$ and block size $= 64$ just obtained previously, Table 5 lists the GPU runtimes for cases with and without coalesced memory access. In the YSU PBL scheme, there are 17 3-D-array variables, 17 2-D-array variables, and two 1-D-array variables needed to be input from CPU to GPUs, amounting to 326 475 352 bytes. For the outputs, this scheme need transfer seven 3-D-array variables and seven 2-D-array variables from GPUs back to the CPU, corresponding to 134 430 912 bytes in total. The data size of the outputs is about 41 % of that of the inputs, which indicates rough consistency with the runtime taken by device-to-host data transfer as compared to that by host-to-device data copy. This is shown in Table 5, where the I/O transfer involved here is synchronous memory access.

This supports the general finding that data transfer between CPU and GPUs takes up a lot of time, and apparently is a limited factor for the speedup. Since the computation of this scheme is only one sub-process of the entire WRF model, and what we are more interested is its speedup with no I/O transfer, for the reasons given in Sect. 4.1.

However, when I/O transfer is considered, the use of asynchronous memory access to overlap CUDA kernel execution with data transfer can be applied. Each Tesla K40 has one copy engine and one kernel engine, allowing data transfer and kernel execution to overlap. As commands pipeline, streams execute commands in a manner of first-in-first-out (FIFO) order. The stream arrangement would result in overlapping CPU-to-GPU and GPU-to-CPU memory transfers and kernel execution on GPUs with two copy engines. A diagram that depicts the execution timeline of the YSU PBL computation process is illustrated in Fig. 11, where the illustrated three streams are in different colors. When asynchronous memory access is taken into account, the results of computing performance are listed at the last row of Table 5. It indicates that using asynchronous access, together with coalesced memory transfer, seems to help reduce the total runtime for this scheme.

4.8 Output comparison between CPU-based and GPU-based programs

It was found that the GPU-based programs of the YSU PBL scheme were sensitive to precision. Two key issues we encountered were: special functions such as *powf(), expf(), sqrtf(),* and the compiler option for the math calculation. Firstly, when C programs were converted to CUDA C programs, the special functions taken from GNU C library source codes must be modified to be used by GPU devices; otherwise incorrect outputs emerge (in comparison to Fortran outputs) if CUDA C built-in special functions were used. Secondly, when the compiler option of math calculation, *"-use_fast_math"* was employed to compile CUDA C programs, incorrect outputs of some variables would appear, again, as compared to Fortran outputs. This compiler option made the CUDA C programs run faster, but it could not produce outputs identical to the Fortran outputs. Thus another math-calculation option *"-fmad=false"* was chosen.

A mathematical function causing the highest unit in the last place (*ulp*) difference between GNU C math library and CUDA C functions is the power function. In Fig. 12, *ulp* differences for power function in GPU CUDA C library with and without fast math option are compared to the GNU C mathematical library. The library function for the power function is *powf*(*x*,*y*). In the plot, *x* starts from value *500.37420654296875* and *y* has a constant value of *0.950593948364257812*5. The spacing of the value *x* in the plot is non-equal as each consecutive value is derived from the previous value by adding one ***ulp*** to the previous value. Thus, the plot shows *ulp* values for 150 consecutive 32-bit floating point values starting from the first value. For these example values, the *ulp* ranges from 0 to 10 with fast math. Without fast math, the maximum *ulp* value is three. Due to error cascading effects in the chaotic YSU PBL algorithm, an *ulp* error of three is capable of causing a large error in the final output. In order to get exactly the same results for the math functions on GPU and CPU, we adopted GNU C math libraries for GPU execution by adding CUDA C device execution qualifiers to the existing C source code.

The root-mean square error (RMSE) is employed to make a comparison between the CUDA C outputs of a variable and Fortran outputs of the same variable by aggregating over all grid-point elements of this variable. The comparison of the maximal RMSE for those variables having different CUDA C outputs from Fortran outputs are listed in Table 6a. In addition, as mentioned above, the fast math compiler option, *–use_fast_math*, may allow us to execute much faster, but it produces less accurate results. Furthermore, the built-in GPU CUDA C special functions are also able to make programs run faster, but again with less accuracy. Table 6b lists the comparison of the speedups for four different running situations. From these studies, it is found that the option *"-fmad=false"* along with our own modified special functions can make our CUDA C programs produce outputs identical to Fortran outputs, albeit with less speedup.

4.9 Multi-GPU implementation

Our multi-GPU setup is displayed in Fig. 13. In this setup, there are two Tesla K40 GPU cards, each of which has one GPU. Multi-GPU implementation of the YSU PBL scheme is performed by computing several contiguous *j*-dimensional values in the arrays within the same GPU. With asynchronous access taken into account, the optimal numbers of *j* dimensions for transfers between CPU and GPUs were found to be 26 for use of one or two GPUs.

Using the optimal block size $= 64$ and registers per thread $= 63$, the YSU PBL scheme was executed on our multi GPUs setup. Table 7 lists the computation times and speedups for single GPU and two GPUs with/without I/O transfer and with/without coalesced memory access. We also ran the Fortran programs using one CPU socket (4 cores), and the runtime and speedup are also listed in the same table.

5 Summary and future work

In this paper, we develop an efficient parallel GPU-based YSU PBL scheme of the WRF model using NVIDIA Tesla K40 GPUs. From our study, the communication bandwidth of data transfer is one of the main limiting factors for computing performance of CUDA codes on GPUs. This limitation holds true for other WRF schemes as well. To ameliorate this problem, several crucial code changes made to improve the computing performance. For example, we launched L1 cache with more memory than shared memory; some temporary arrays have been scalarized; and the dependence among the vertical-level components has been freed. In addition, the effects of threads per block and registers per thread on the GPU-based YSU PBL scheme were studied. We also discussed how the compiler options for math calculation would affect the outputs of the GPU-based programs. At the end, we came up with an optimized GPU-based YSU PBL scheme with outputs identical to the CPU-based Fortran programs.

When WRF is fully implemented on GPUs, the implementation of input/output transfers between CPU and GPU(s) will not be needed for each intermediate module. Instead, only inputs to the first WRF module have to be transferred from CPU to GPU(s), and only the outputs from the last WRF module will be transferred from GPU(s) to CPU. The YSU PBL scheme is only one intermediate module of the entire WRF model. Therefore, the speedups for the YSU PBL scheme are expected to be close to the results presented in the cases without I/O transfer rather than to those with I/O transfer.

Using one NVIDIA Tesla K40 GPU in the case without I/O transfer, our optimization efforts on the GPU-based YSU PBL scheme can achieve a speedup of $193\times$ with respect to one CPU core, whereas the speedup for one CPU socket (4 cores) with respect to one CPU core is only $3.5\times$. We also ran the CPU-based code on one CPU core using exactly the same optimization along with height dependence release as the GPU-based code, and its speedup is merely 1.5x as compared to its original Fortran counterpart. In addition, we can even boost the GPU-based speedup to $360\times$ with respect to one CPU core when two K40 GPUs are applied; in this case, one minute of model execution on dual Tesla K40 GPUs will achieve the same outcome as six hours of execution on a single core CPU.

Our future work is to continue accelerating other parts of WRF model using GPUs. Eventually, we expect to have a WRF model completely running on GPUs. This will provide a superior performance for weather research and forecasting in the near future.

Acknowledgements. This work is supported by the National Oceanic and Atmospheric Administration (NOAA) under Grant No. NA10NES4400013. B. Huang would like to thank Stanley Posey of NVIDIA for donating Tesla K40 GPUs. The authors are also grateful to James E. Davies of Space Science and Engineering Center at University of Wisconsin-Madison for professional editing of this manuscript.

References

Betts, A., Hong, S.-Y., and Pan, H.-L.: Comparison of NCEPNCAR reanalysis with 1987 FIFE data, Mon. Weather Rev., 124, 1480–1498, 1996.

Bright, D. R. and Mullen, S. L.: The sensitivity of the numerical simulation of the southwest monsoon boundary layer to the choice of PBL turbulence parameterization in MM5, Weather Forecast., 17, 99–114, 2002.

Continental US (CONUS): WRF V3 Parallel Benchmark Page, Single domain, medium size, 12 km CONUS, Oct. 2001, available at: http://www2.mmm.ucar.edu/wrf/WG2/benchv3/#_Toc212961288, 18 June 2008.

Durran, D. R. and Klemp, J. B.: The effects of moisture on trapped mountain lee waves, J. Atmos. Sci., 39, 2490–2506, 1982.

Hong, S.-Y. and Pan, H.-L.: Nonlocal boundary layer vertical diffusion in a Medium-Range Forecast model, Mon. Weather Rev., 124, 2322–2339, 1996.

Hong, S.-Y., Noh, Y., and Dudhia, J.: A new vertical diffusion package with an explicit treatment of entrainment processes, Mon. Weather Rev., 134, 2318–2341, 2006.

Horn, S.: ASAMgpu V1.0 – a moist fully compressible atmospheric model using graphics processing units (GPUs), Geosci. Model Dev., 5, 345–353, doi:10.5194/gmd-5-345-2012, 2012.

Kim, J. and Mahrt, L.: Simple formulation of turbulent mixing in the stable free atmosphere and nocturnal boundary layer, Tellus A, 44, 381–394, 1992.

Klingbeil, G., Erban, R., Giles, M., and Maini, P. K.: Fat versus thin threading approach on GPUs: application to stochastic simulation of chemical reactions, IEEE T. Parall. Distr., 23, 2, 280–287, doi:10.1109/TPDS.2011.157, 2012.

Louis, J. F.: A parametric model of vertical eddy fluxes in the atmosphere, Bound.-Lay. Meteorol., 17, 187–202, 1979.

Mielikainen, J., Huang, B., Huang, H.-L. A., Goldberg, M. D., and Mehta, A.: Speeding up the computation of WRF double moment 6-class microphysics scheme with GPU, J. Atmos. Ocean. Tech., vol 30, 2896–2906, doi:10.1175/JTECH-D-12-00218.1, 2013.

Moeng, C. H. and Sullivan, P. P.: A comparison of shear and buoyancy-driven planetary boundary layer flows, J. Atmos. Sci., 51, 999–1022, 1994.

Noh, Y., Cheon, W. G., Hong, S.-Y., and Raasch, S.: Improvement of the K-profile model for the planetary boundary layer based on large eddy simulation data, Bound.-Lay. Meteorol., 107, 401–427, 2003.

NVIDIA: Tesla GPU Accelerators, available at: http://www.nvidia.com/content/tesla/pdf/NVIDIA-Tesla-Kepler-Family-Datasheet.pdf, October 2013.

NVIDIA: Tesla K40 versus K20 GPU, available at: http://blog.xcelerit.com/benchmarks-nvidia-tesla-k40-vs-k20x-gpu/, November 2013.

Overview of WRF Physics: available at: http://www2.mmm.ucar.edu/wrf/users/tutorial/201301/dudhia_physics.pdf, April 2013.

Paz, A. and Plaza, A.: Clusters versus GPUs for parallel automatic target detection in remotely sensed hyperspectral images, EURASIP J. Adv. Sig. Pr., 35, 18 pp., doi:10.1155/2010/915639, 2010.

Price, E., Mielikainen, J., Huang, M., Huang, B., Huang, H.-L. A., and Lee, T.: GPU-Accelerated Longwave Radiation Scheme of the Rapid Radiative Transfer Model for General Circulation Models (RRTMG), IEEE J. Sel. Top. Appl. Earth Observ. Remote Sens. (JSTARS), 7, 8, 3660–3667, doi:10.1109/JSTARS.2014.2315771, 2014.

Rustico, E., Bilotta, G., Herault, A., Negro, C. D., and Gallo, G.: Advances in Multi-GPU Smoothed Particle Hydrodynamics Simulations, IEEE T. Parall. Distr., 25, 1, 43–52, doi:10.1109/TPDS.2012.340, 2014.

Siewertsen, E., Piwonski, J., and Slawig, T.: Porting marine ecosystem model spin-up using transport matrices to GPUs, Geosci. Model Dev., 6, 17–28, doi:10.5194/gmd-6-17-2013, 2013.

Song, C., Li, Y., and Huang, B.: A GPU-accelerated wavelet decompression system with SPIHT and Reed-Solomon decoding for satellite images, IEEE J. Sel. Topics Appl. Earth Observ. Remote Sens. (JSTARS), 4, 683–690, doi:10.1109/JSTARS.2011.2159962, 2011.

Troen, I. and Mahrt, L.: A simple model of the atmospheric boundary layer sensitivity to surface evaporation, Bound.-Lay. Meteorol., 37, 129–148, 1986.

Wei, S.-C. and Huang, B.: GPU acceleration of predictive partitioned vector quantization for ultraspectral sounder data compression, IEEE J. Sel. Top. Appl., 4, 677–682, 2011.

Wu, X., Huang, B., Plaza, A., Li, Y., and Wu, C.: Real-time implementation of the pixel purity index algorithm for endmember identification on GPUs, IEEE Geosci. Remote S., 11, 955–959, 2014.

Table 1. GPU runtime and speedup as compared to one single-threaded CPU core for the first CUDA C version of the YSU PBL scheme, where block size = 64 and registers per thread = 63 are used.

	CPU runtime	GPU runtime	Speedup
One CPU core	1800.0 ms		
Non-coalesced		36.0 ms	50.0×
Coalesced		34.2 ms	52.6×

Table 2. GPU runtime and speedup as compared to one single-threaded CPU core for the first CUDA C version of the YSU PBL scheme after the L1 cache command "cudaFuncCachePreferL1" is applied, where block size = 64 and registers per thread = 63 are used.

	CPU runtime	GPU runtime	Speedup
One CPU core	1800.0 ms		
Non-coalesced		34.3 ms	$52.5\times$
Coalesced		33.0 ms	$54.5\times$

Table 3. GPU runtime and speedup as compared to one single-threaded CPU core after further improvement with replacing temporary arrays by scalar variables, where block size = 64 and registers per thread = 63 are used.

	CPU runtime	GPU runtime	Speedup
One CPU core	1800.0 ms		
Non-coalesced		22.3 ms	$80.7\times$
Coalesced		21.4 ms	$84.1\times$

Table 4. GPU runtime and speedup as compared to one single-threaded CPU core after releasing height dependence, where block size $= 64$ and registers per thread $= 63$ are used.

	CPU runtime	GPU runtime	Speedup
One CPU core	1800.0 ms		
Non-coalesced		10.74 ms	$167.6\times$
Coalesced		9.29 ms	$193.8\times$

Table 5. GPU runtime and speedup as compared to one single-threaded CPU core for data transfer between CPU and GPIUs, where block size $= 64$ and registers/thread $= 63$ are used.

CPU runtime	1800.0 ms			
GPU runtime	Non-coalesced	Speedup	Coalesced	Speedup
host to device	27.26 ms		30.69 ms	
kernel execution	10.74 ms	$167.6\times$	9.29 ms	$193.8\times$
device to host	10.83 ms		20.39 ms	
kernel $+$ Sync I/O	48.83 ms	$36.9\times$	60.37 ms	$29.8\times$
with Async I/O	52.55 ms	$34.3\times$	54.66 ms	$32.9\times$

Table 6a. Comparison of maximal RMSE for those variables having different outputs from Fortran outputs for four different combination cases, where "Spec" stands for special function.

Variables	A	B	C	D
Horizontal velocity u component	2.5e-04	1.1e-08	4.1e-06	0
Horizontal velocity v component	2.3e-04	1.0e-08	1.8e-06	0
Potential temperature (θ)	5.2e-05	1.0e-07	1.1e-06	0
Mixing ratio of water vapor (q_v)	4.3e-08	1.3e-11	1.5e-09	0
Mixing ratio of cloud water (q_c)	1.3e-08	4.7e-14	1.9e-12	0
Exchange coefficient	3.0e-00	2.9e-06	1.8e-03	0
PBL height	3.9e+02	1.3e-04	2.8e-02	0

A: -use_fast_math+ CUDA C built-in Spec;
B: -fmad=false + CUDA C built-in Spec;
C: -use_fast_math + our own Spec;
D: -fmad=false + our own Spec.

Table 6b. Comparisons of speedups using one GPU for four different combination cases, where "Spec" stands for special functions. Note that the optimized GPU-based YSU PBL scheme along with block size = 64 and register/thread = 63 is used in these comparison.

GPU-Based code running conditions	A	B	C	D
Non-coalesced	208.9×	151.6×	193.1×	167.6×
Coalesced	311.2×	167.7×	250.3×	193.8×

A: -use_fast_math+ CUDA C built-in Spec;
B: -fmad=false + CUDA C built-in Spec;
C: -use_fast_math + our own Spec;
D: -fmad=false + our own Spec.

Table 7. Results of runtime and speedup for CPU-based and optimized GPU-based YSU PBL scheme, where block size = 64 and registers per thread = 63, along with compiler option "-fmad=false" as well as our own modified special functions, are used.

	CPU runtime		Speedup	
One CPU core	1800.0 ms			
One CPU socket	509.0 ms		3.5×	
	GPU runtime		Speedup	
Without I/O	Non-coalesced	Coalesced	Non-coalesced	Coalesced
1 GPU	10.74 ms	9.29 ms	167.6×	193.8×
2 GPUs	5.80 ms	4.99 ms	310.9×	360.6×
	GPU runtime		Speedup	
Async I/O	Non-coalesced	Coalesced	Non-coalesced	Coalesced
1 GPU	52.55 ms	54.66 ms	34.3×	32.9×
2 GPUs	35.88 ms	46.31 ms	50.2×	38.9×

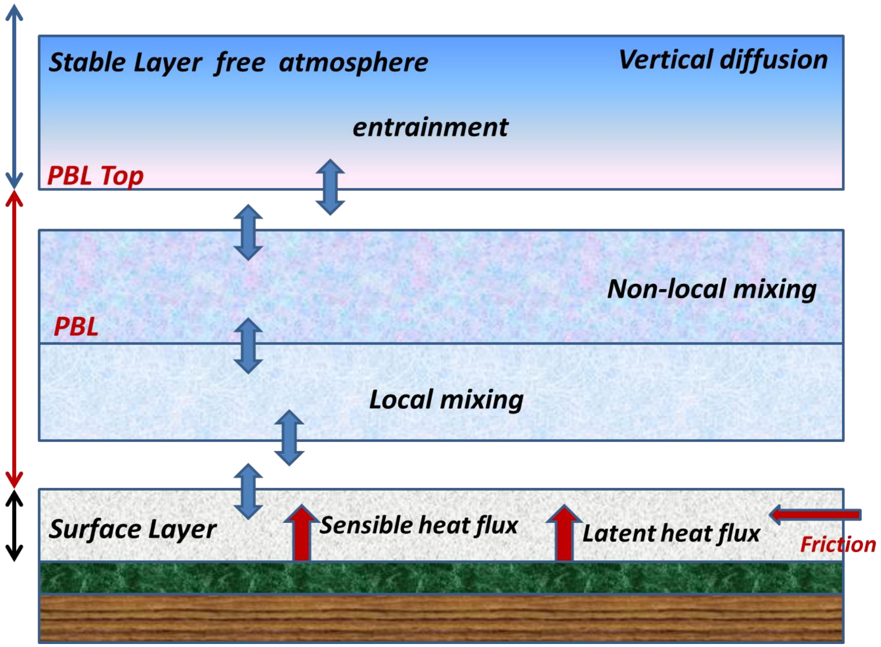


Figure 1. Illustration of PBL process (Overview of WRF Physics).

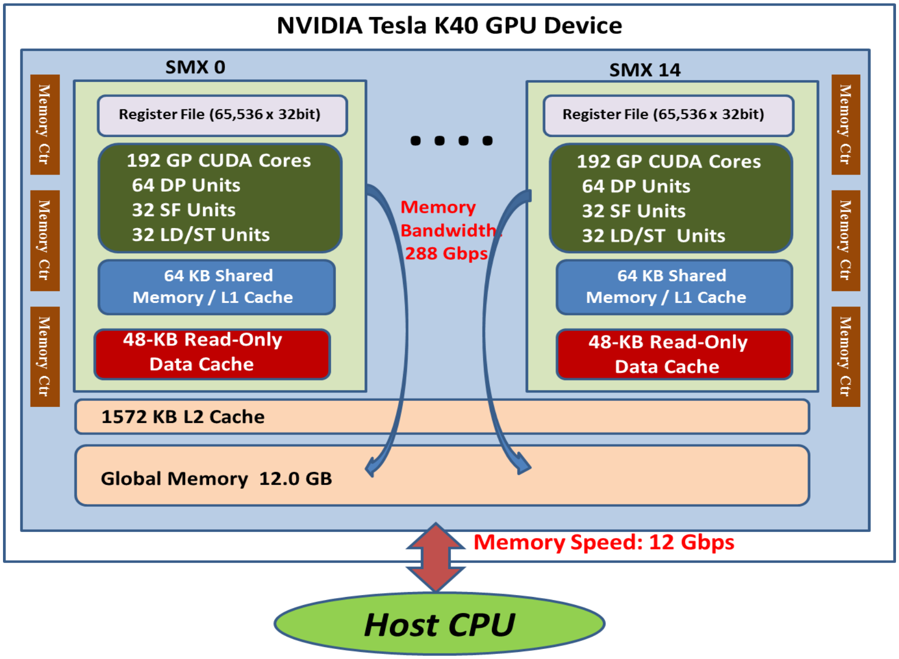


Figure 2. Schematic diagram of hardware specification of one NVIDIA Tesla K40 GPU employed in our study.

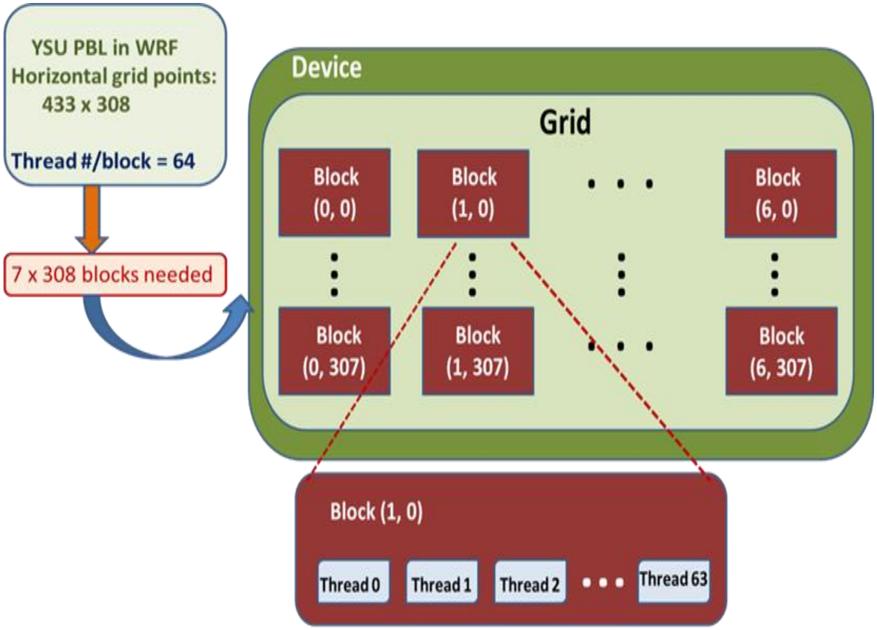


Figure 3. Three-level thread hierarchy of a device for one GPU utilized in our study: threads, thread block, and grids of block.

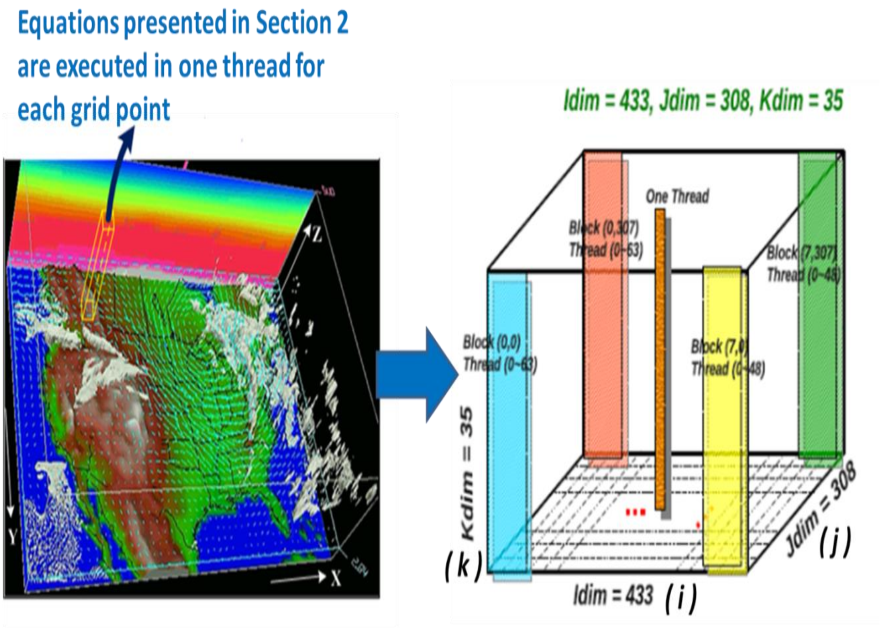

Equations presented in Section 2 are executed in one thread for each grid point
Idim = 433, Jdim = 308, Kdim = 35
One Thread
Block (0,307) Thread (0~63)
Block (7,307) Thread (0~48)
Block (0,0) Thread (0~63)
Block (7,0) Thread (0~48)
Kdim = 35
(k)
Idim = 433 (i)
Jdim = 308
(j)
Z
Y
X

Figure 4. Mapping of the CONUS domain onto one GPU thread-block-grid domain, where the size of the CONUS domain is 433×308 horizontal grid point with 35 vertical levels.

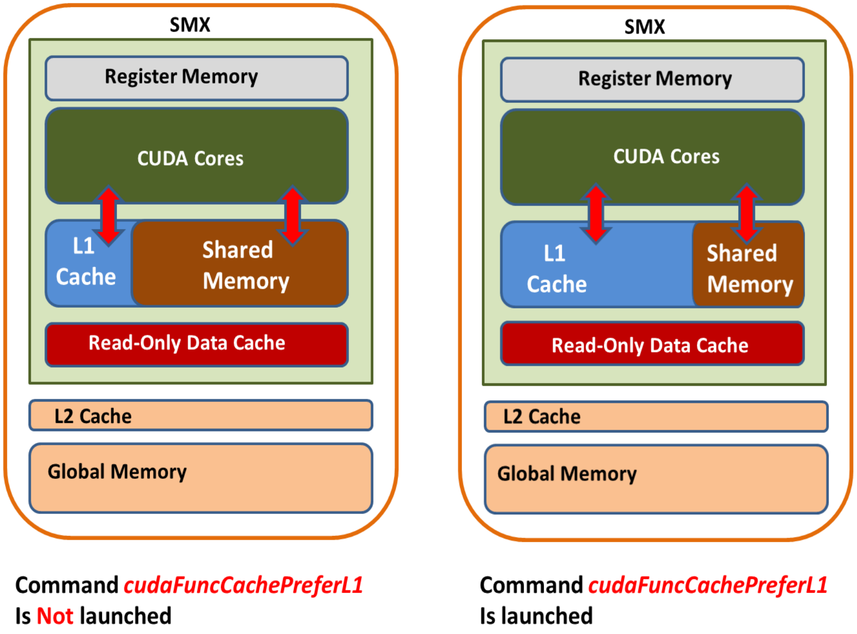


Figure 5. Memory allocation of L1 cache and shared memory by launching "*cudaFuncCachePreferL1*" or not.

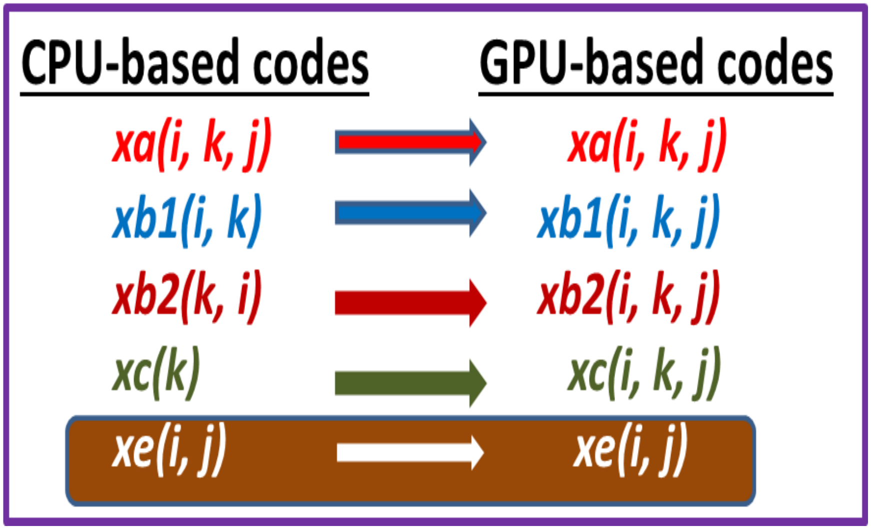
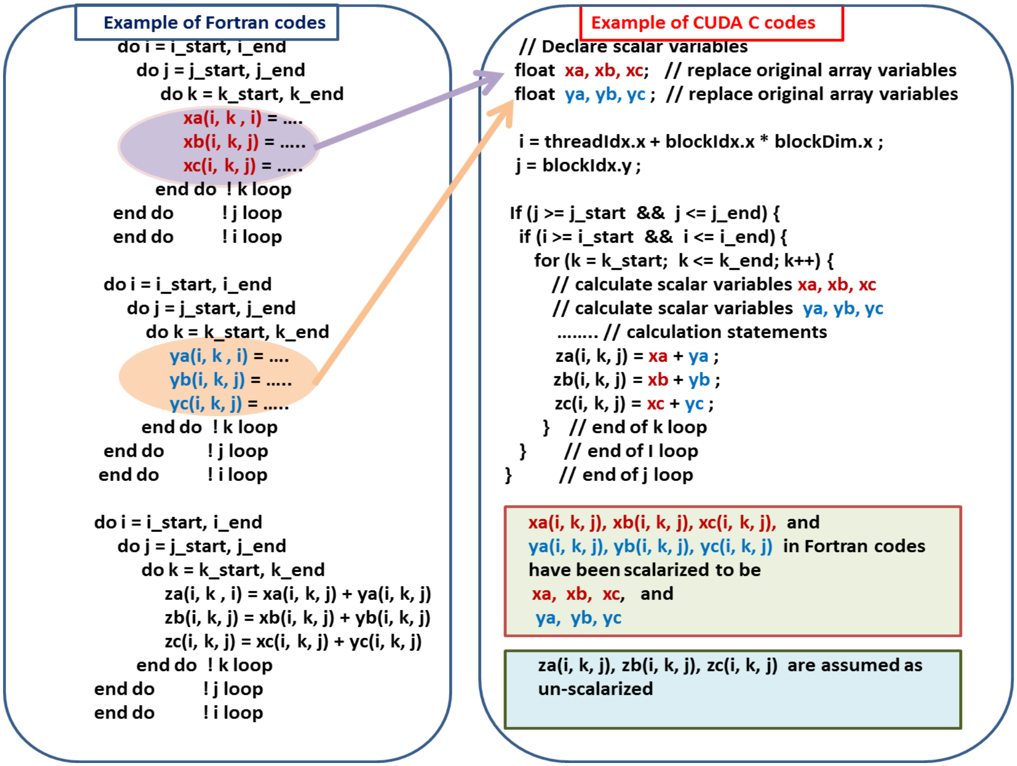


Figure 6. Illustration for re-arranging arrays in CPU-based implementation to 3-D and 2-D arrays in GPU-based implementation, where regular mathematical array syntax is used to express the arrays.



Example of Fortran codes

```
do i = i_start, i_end
   do j = j_start, j_end
      do k = k_start, k_end
           xa(i, k , i) = ....
           xb(i, k, j) = .....
           xc(i, k, j) = .....
      end do  ! k loop
end do        ! j loop
end do        ! i loop

do i = i_start, i_end
   do j = j_start, j_end
      do k = k_start, k_end
           ya(i, k , i) = ....
           yb(i, k, j) = .....
           yc(i, k, j) = .....
      end do  ! k loop
end do        ! j loop
end do        ! i loop

do i = i_start, i_end
   do j = j_start, j_end
      do k = k_start, k_end
         za(i, k , i) = xa(i, k, j) + ya(i, k, j)
         zb(i, k, j) = xb(i, k, j) + yb(i, k, j)
         zc(i, k, j) = xc(i, k, j) + yc(i, k, j)
      end do  ! k loop
end do        ! j loop
end do        ! i loop
```

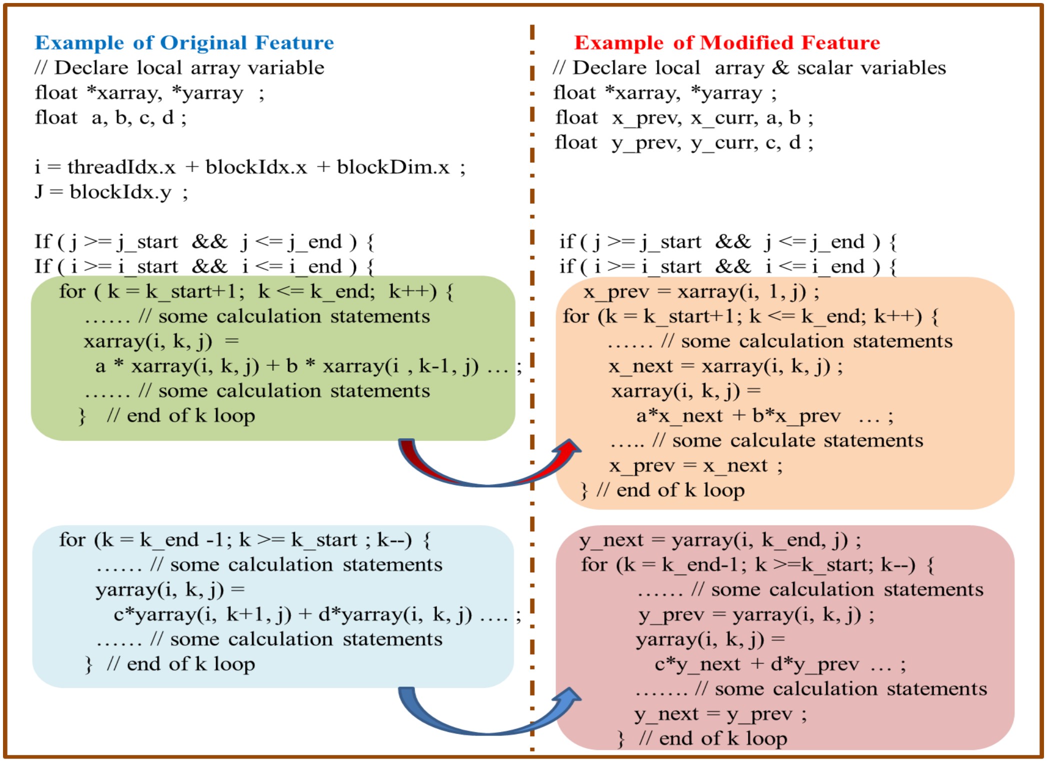
Example of CUDA C codes

```
// Declare scalar variables
float  xa, xb, xc;   // replace original array variables
float  ya, yb, yc ;  // replace original array variables

 i = threadIdx.x + blockIdx.x * blockDim.x ;
 j = blockIdx.y ;

If (j >= j_start  &&  j <= j_end) {
 if (i >= i_start  &&  i <= i_end) {
   for (k = k_start;  k <= k_end; k++) {
     // calculate scalar variables xa, xb, xc
     // calculate scalar variables  ya, yb, yc
     ........ // calculation statements
     za(i, k, j) = xa + ya ;
     zb(i, k, j) = xb + yb ;
     zc(i, k, j) = xc + yc ;
   }  // end of k loop
 }    // end of I loop
}     // end of j loop
```

xa(i, k, j), xb(i, k, j), xc(i, k, j), and
ya(i, k, j), yb(i, k, j), yc(i, k, j) in Fortran codes
have been scalarized to be
xa, xb, xc, and
ya, yb, yc

za(i, k, j), zb(i, k, j), zc(i, k, j) are assumed as un-scalarized

Figure 7. Illustration of scalarizing temporary arrays by scalar variable in order to recalculate values in the local memory rather than transferring them from global memory to local memory. In this illustration , *xa, xb, xc, ya, yb, yc* are assumed to be scalarable, but *za, zb, zc* are assumed to be un-scalarizable. Note that the regular mathematical array syntax is used to express the arrays.



Example of Original Feature

```
// Declare local array variable
float *xarray, *yarray  ;
float  a, b, c, d ;

i = threadIdx.x  + blockIdx.x  + blockDim.x  ;
J = blockIdx.y  ;

If ( j >= j_start   &&  j <= j_end ) {
If ( i >= i_start   &&  i <= i_end ) {
   for ( k = k_start+1;  k <= k_end;  k++) {
      ...... // some calculation statements
      xarray(i, k, j)  =
        a * xarray(i, k, j) + b * xarray(i , k-1, j) ... ;
      ...... // some calculation statements
     }   // end of k loop

   for (k = k_end -1; k >= k_start ; k--)  {
       ...... // some calculation statements
       yarray(i, k, j) =
          c*yarray(i,  k+1, j) + d*yarray(i,  k, j) .... ;
       ...... // some calculation statements
     }  // end of k loop
```

Example of Modified Feature

```
// Declare local  array & scalar variables
float *xarray, *yarray ;
float  x_prev, x_curr, a, b ;
float  y_prev, y_curr, c, d ;

if ( j >= j_start   &&  j <= j_end ) {
if ( i >= i_start   &&  i <= i_end ) {
   x_prev = xarray(i, 1, j) ;
 for (k = k_start+1; k <= k_end; k++) {
       ...... // some calculation statements
       x_next = xarray(i, k, j) ;
        xarray(i, k, j) =
            a*x_next + b*x_prev   ... ;
       ..... // some calculate statements
       x_prev = x_next ;
  } // end of k loop

   y_next = yarray(i, k_end, j) ;
   for (k = k_end-1; k >=k_start; k--)  {
          ...... // some calculation statements
          y_prev = yarray(i, k, j) ;
          yarray(i, k, j) =
             c*y_next + d*y_prev  ... ;
          ....... // some calculation statements
          y_next = y_prev ;
      }  // end of k loop
```

Figure 8. Illustration for how to release height dependence in order to reduce the access time to global memory for those temporary arrays which have dependence among the $(k-1)$th, kth, and $(k+1)$th components. The left (right) figure is the original (modified) CUDA program.

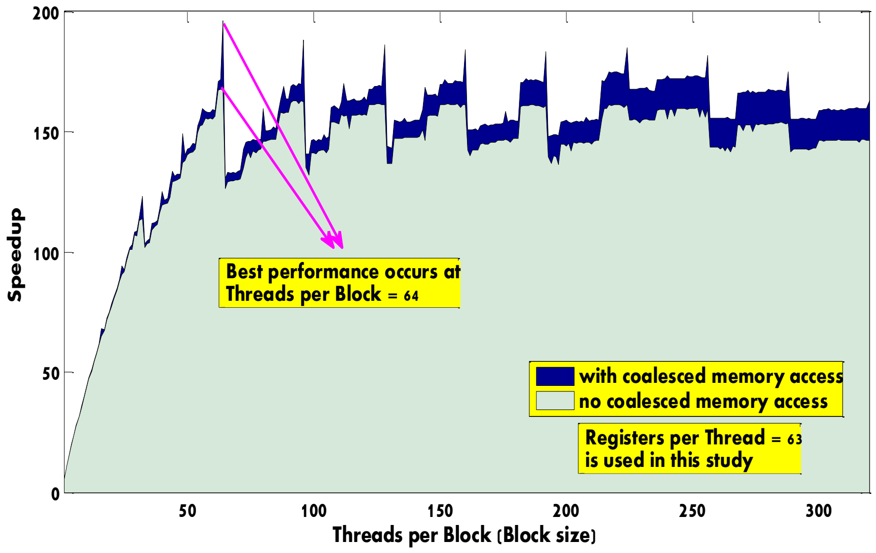


Figure 9. Speedup vs. threads per block (block size) for optimized GPU-based YSU PBL scheme.

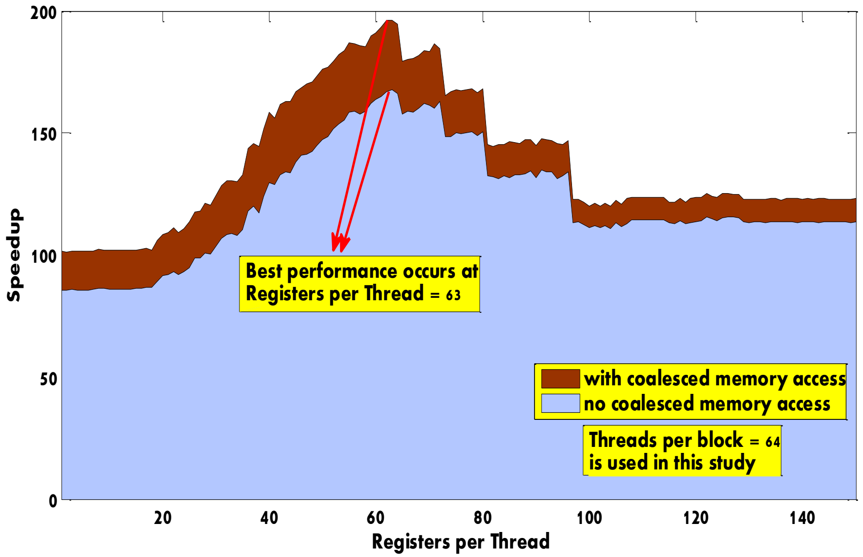


Figure 10. Speedup vs. registers per thread for optimized GPU-based YSU PBL scheme.

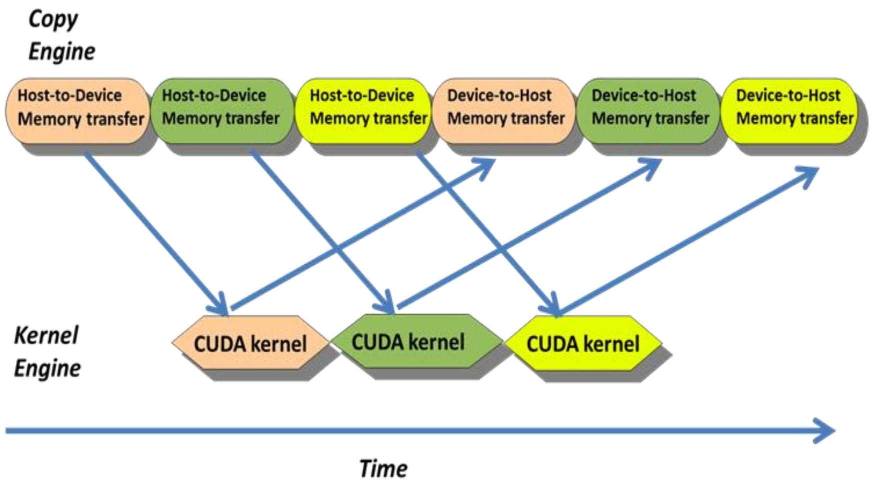

Copy Engine
Host-to-Device Memory transfer
Host-to-Device Memory transfer
Host-to-Device Memory transfer
Device-to-Host Memory transfer
Device-to-Host Memory transfer
Device-to-Host Memory transfer
Kernel Engine
CUDA kernel
CUDA kernel
CUDA kernel
Time

Figure 11. Asynchronous memory transfer among host-to-device memory transfer, GPU kernel execution, and device-to-host memory transfer.

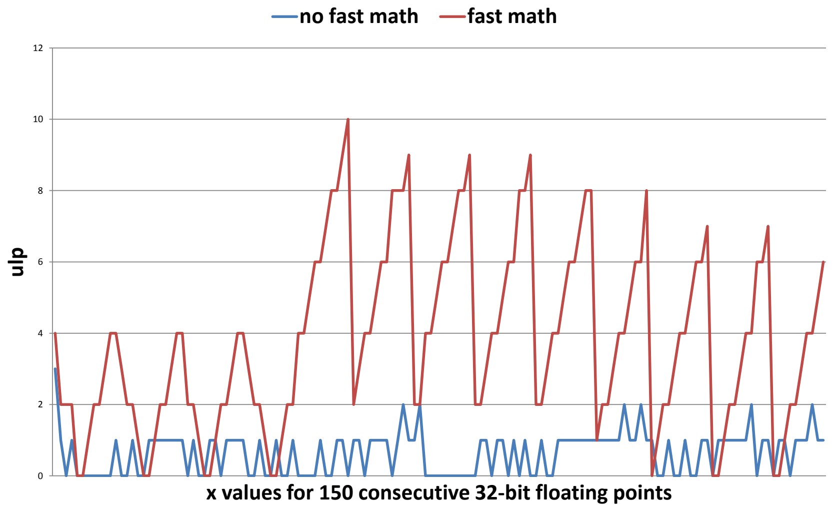


Figure 12. Unit in the last place error for power function. Both fast math and no fast math GPU options are compared to the GNU C math library.

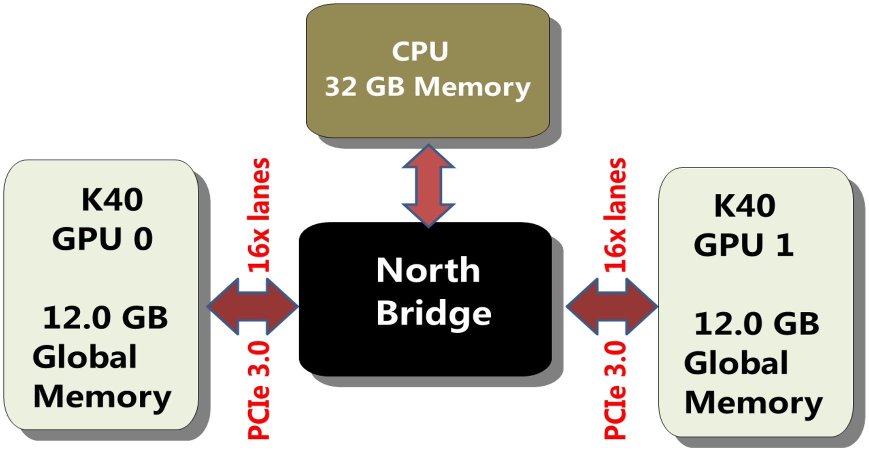


Figure 13. Schematic diagram of our multi-GPU setup.