Advances in Engineering Software 115 (2018) 248–267

Contents lists available at ScienceDirect

Advances in Engineering Software

journal homepage: www.elsevier.com/locate/advengsoft

Research paper

Graphics processing unit accelerated phase field dislocation dynamics: Application to bi-metallic interfaces

Adnan Eghtesad[a], Kai Germaschewski[b], Irene J. Beyerlein[c], Abigail Hunter[d], Marko Knezevic[a,*]

[a] Department of Mechanical Engineering, University of New Hampshire, Durham, NH 03824, USA
[b] Department of Physics, University of New Hampshire, Durham, NH 03824, USA
[c] Mechanical Engineering Department, Materials Department, University of California Santa Barbara, Santa Barbara, CA 93106-5070, USA
[d] X Computational Physics Division, Los Alamos National Laboratory, Los Alamos, NM 87544, USA

ARTICLE INFO

Keywords:
Phase field dislocation dynamics
Graphics processing unit (GPU)
Compute unified device architecture (CUDA)
OPENACC
Spectral methods

ABSTRACT

We present the first high-performance computing implementation of the meso-scale phase field dislocation dynamics (PFDD) model on a graphics processing unit (GPU)-based platform. The implementation takes advantage of the portable OpenACC standard directive pragmas along with Nvidia's compute unified device architecture (CUDA) fast Fourier transform (FFT) library called CUFFT to execute the FFT computations within the PFDD formulation on the same GPU platform. The overall implementation is termed ACCPFDD-CUFFT. The package is entirely performance portable due to the use of OPENACC-CUDA inter-operability, in which calls to CUDA functions are replaced with the OPENACC data regions for a host central processing unit (CPU) and device (GPU). A comprehensive benchmark study has been conducted, which compares a number of FFT routines, the Numerical Recipes FFT (FOURN), Fastest Fourier Transform in the West (FFTW), and the CUFFT. The last one exploits the advantages of the GPU hardware for FFT calculations. The novel ACCPFDD-CUFFT implementation is verified using the analytical solutions for the stress field around an infinite edge dislocation and subsequently applied to simulate the interaction and motion of dislocations through a bi-phase copper-nickel (Cu–Ni) interface. It is demonstrated that the ACCPFDD-CUFFT implementation on a single TESLA K80 GPU offers a 27.6X speedup relative to the serial version and a 5X speedup relative to the 22-multicore Intel Xeon CPU E5-2699 v4 @ 2.20 GHz version of the code.

1. Introduction

The study of the mechanical behavior of polycrystalline materials with a grain size of typically less than 0.1 μm has been the focus of materials researchers for several decades [1–9] In polycrystalline materials, dislocations are line defects whose motion carries plastic deformation. The atomic core structure of a dislocation deviates from the perfect atomic structure of the crystal. Understanding the deformation mechanisms of moving dislocations and their interplay with interfaces and grain boundaries is crucial in understanding polycrystalline plasticity.

In order to investigate the multiscale nature of plasticity in metals, several modeling approaches encompassing different length scales have been developed. Molecular dynamics (MD) and density functional theory (DFT) are powerful techniques to capture the motion and configuration of dislocations at an atomistic level. While MD and DFT simulations are promising tools for capturing dislocation motion and interactions, they are computationally expensive due to the need to resolve the entire atomic structure limiting the time and length scales that can be achieved in simulations. In MD, for instance, the grain size limitation is approximately 100 nm [10]. To relax the computational speed concerns, MD models have already been implemented on graphics processing unit (GPU) hardware [11,12]. Nevertheless, a mesoscale technique is desirable to connect length and time scales that are more consistent with experimental scales. Generally, in order to study the dislocation motion and configuration in length scales beyond order of microns, mesoscale methods are implemented. Discrete dislocation dynamics (DDD) and phase field methods are two major categories for such studies, both of which resolve individual dislocations rather than the atomic structure of a material. DDD simulations model individual dislocations by dividing each dislocation line into a series of line segments connected with nodes. The dislocation structure is then evolved through the calculation of elastic forces on each dislocation line segment and then by solving the equation of motions. The forces on each dislocation are determined as the summation of stress magnitudes over all of the dislocation line segments. In most cases, isotropic elasticity

* Corresponding author.
E-mail address: marko.knezevic@unh.edu (M. Knezevic).

http://dx.doi.org/10.1016/j.advengsoft.2017.09.010
Received 18 July 2017; Received in revised form 12 September 2017; Accepted 27 September 2017
Available online 14 October 2017

theory is assumed [13–18].

On the other hand, a phase field formulation is a modeling approach that operates on a similar length as DDD; however, each individual dislocation is resolved using order parameters or phase field variables. Each order parameter is a scalar valued variable that tracks areas slipped by dislocations and for every point counts how many dislocations have traversed. Order parameters (and hence the dislocation structure) are evolved through minimization of the total system energy. One of the energy terms, the strain energy, relies on the calculation of a complex interaction matrix, which will be described shortly. Through transformation to Fourier space, the calculation of this matrix is greatly simplified, but it must then utilize periodic boundary conditions and fast Fourier transforms (FFT) computations. In this work, we have chosen to focus on a phase field code called phase field dislocation dynamics (PFDD) [19].

Despite the fact that the PFDD model accounts for larger scales in comparison to MD simulations, study of a large problem size including massive number of FFT voxels can still result in computationally expensive simulations. For example, in a face-centered cubic (fcc) system, a PFDD simulation can have up to 12 active order parameters due to the existence of 12 slip systems resulting from the crystal symmetry and the atomically close-packed planes. In order to fully evolve the plasticity, these 12 slip systems can be taken into count with 12 phase field variables. More than that, for a bicrystal interface system we must also evolve the virtual strains due to the second phase, resulting in a 3×3 system of equations that adds an additional 9 order parameters, one for each component of the virtual strain. The total energy of the system must be minimized with respect to each active order parameter. In an fcc material, this can translate into up to 12 coupled integro-differential equations that must be solved. Furthermore, as more complex physics are incorporated into the PFDD model, computing complexity also increases. Recently, the PFDD model was extended to account for bi-phase interfaces [20]. In this case, there is an additional strain tensor (the virtual strain) incorporated to account of material inhomogeneities. This strain field is dependent upon the current dislocation configuration, and hence must also be evolved through minimization of the total system energy. This adds an additional nine equations to those done for the order parameters that must be solved in order to determine the equilibrium state of the system. The additional computation time required for a PFDD simulation involving multi-dislocation-interface set-ups makes such simulations impractical to execute at a reasonable wall clock time using a serial framework. Consequently, a critical need arises to accelerate the PFDD simulations.

The total number of cores used in a single central processing unit (CPU) is limited by the CPU hardware. While modern CPUs utilize more cores and wider single instruction multiple data (SIMD) units, high performance super computers made up of merely CPUs can be highly expensive to operate. On the other hand, with the advent of graphics processing units (GPUs), running large-scale simulations have become more feasible than ever before. GPUs are accelerators, originally developed for 3D visualizations and optimized for parallel processing of millions of polygons with very large datasets [21]. GPUs are considerably faster in comparison to CPUs for processing large datasets. For example, the memory bandwidth on Nvidia's Tesla K80 GPU is up to 480 GB/sec, while it is no more than 68 GB/sec for systems with PC3-17000 DDR3 modules and quad-channel architecture. In terms of computational power, Tesla K80 is capable of achieving up to 2.91 and 8.74 TFLOPS for double precision and single precision, respectively, while it is no more than 900 GFLOPS for an Intel Xeon CPU E5-2699 v4 @ 2.20 GHz with 22 cores, when using AVX2 and FMA3 instructions with the turbo boost enabled [22]. Furthermore, GPUs are much cheaper than CPUs, and a decent GPU can turn a simple desktop into a high performance cluster running thousands of concurrent GPU threads. Last but not least, GPU-ported applications are growing very fast to address industrial and academic demands for massively parallel metal forming simulations. Examples include finite element software packages [23,24] and polycrystal plasticity models [25–29].

In this work, we present the first parallel implementation of the PFDD model on GPUs, which we will refer to as ACCPFDD. The ACCPFDD implementation takes advantage of the OPENACC standard directive pragmas and the compute unified device architecture (CUDA) FFT library (CUFFT) and is thus termed ACCPFDD-CUFFT along with OPENACC-CUDA interoperability for efficient acceleration on a single Tesla K80 GPU. The implementation offers scalability of a 27.6X speedup compared to the serial PFDD code and a 5X speedup compared to running a simulation on a 22-Multicore Intel Xeon E5 2699v4 CPU. This speedup is obtained for a problem size of 524,288 FFT points and can improve further with increasing domain size (i.e. total FFT sampling points). The combination of OpenACC with CUDA using the cutting-edge interoperability has been performed over several levels throughout the whole code (i.e. intensive energy computations and FFT calculations) in order to accelerate the computations, while maintaining the GPU program "performance portable" for future studies.

This paper is organized as follows: Section 2 summarizes the main PFDD governing equations for a bi-phase interface, and validation and application case studies are described in Section 3. Section 4 investigates the PFDD hotspots and parallelization steps on GPU. In Section 5, a comprehensive study of potential standard FFT solvers is introduced and discussed in detail. Section 6 explains the final CUPFDD-CUFFT implementation using the OPENACC-CUDA interoperability. Conclusions are presented in Section 7.

2. Summary of basic equations in phase field dislocation dynamics

In phase field algorithms, the physical behavior of a quantity of interest is predicted by tracking and evolving one or more scalar order parameters. These order parameters could be representative of a wide range of phenomena depending on the system of interest such as different crystal structures during phase transformations, fractured regions for crack growth, and different orientations in grain growth models [30]. As mentioned briefly in the introduction, in PFDD the order parameter represents line defects in the crystal lattice called dislocations. The order parameters track the sign and number of glide dislocations on each active slip system in the family $\{111\} < 1\bar{1}0>$, which are atomically closed packed directions in fcc metals. Plasticity in these metals is mediated through the motion and interaction of dislocations. Hence, in PFDD the plastic strain, ε^p is defined as a function of the active order parameters and the Schmid tensor for all slip systems (in our notation, tensors are denoted by bold letters, while tensor components and scalars are shown in italics),

$$\varepsilon_{ij}{}^{p}(\mathbf{x}, t) = \frac{1}{2}\sum_{\alpha=1}^{N} b\xi_{\alpha}(\mathbf{x}, t)\delta_{n}(m_i{}^{\alpha}n_j{}^{\alpha} + m_j{}^{\alpha}n_i{}^{\alpha}), \tag{1}$$

where ξ_α is the order parameter (phase field variable) on slip system α, N is the total number of active slip systems, b is the magnitude of Burgers vector, m^α and n^α are the slip direction and slip plane normal, respectively, and δ_n denotes the Dirac delta function.

As mentioned earlier, the order parameters are evolved through the minimization of the total system energy. In the case of bi-phase interfaces, the total energy is comprised of three key terms [20]

$$E^{total} = E^{strain} + E^{core} + E^{res}, \tag{2}$$

where, E^{strain}, E^{core}, and E^{res} represent the strain energy of the system, dislocation core energy, and the energy to form a residual dislocation in the interface following a slip transmission event, respectively.

In the remainder of this section, we review key equations for the formulation of PFDD for bi-phase interfaces. A more detailed discussion of the formulation can be found in [20,31]. In addition, PFDD has also been used to study perfect and partial dislocation motion in nano-sized grains in fcc metals without the presence of interfaces. Details of this formulation can be found in [19,31,32].

2.1. The strain energy for bi-phase materials

The strain energy accounts for both short-range and long-range internal interactions between dislocations themselves in addition to the interaction with the external applied stress. Additionally, composites, polycrystalline metals, materials with voids and cracks, bi-phase material interfaces and nano-layers introduce a type of heterogeneity due to differences in the elastic moduli resulting in the presences of image forces or Koehler forces that can affect dislocation behavior at interfaces and grain boundaries. The formulation of the strain energy for a bi-phase material in PFDD is based on the Eshelby inclusion method [33]. The system is described as a homogeneous matrix medium containing inclusions or inhomogeneities. The solution is then obtained by replacing the inclusions in the matrix with eigen (virtual) strains due to the presence of the inhomogeneities. In the case of slip transmission through a bi-phase material, the second or recipient material stands for the inhomogeneity relative to the first or donor material, which is both where the dislocation originates and the homogenous matrix.

For the bi-phase copper-nickel (Cu–Ni) system we have strains due to both plasticity and the difference in elastic moduli mismatch between phases 1 and 2, which we denote as a virtual strain. The virtual strain is formulated based on the concept of an Eshelby inhomogeneity [34] by treating material one as the matrix and material two as an inhomogeneity due to its differing elastic modulus. This together leads to the following definition of the strain measure:

$$\varepsilon^0_{ij}(\mathbf{x}) = \begin{cases} \varepsilon^p_{ij}(\mathbf{x}) & \mathbf{x} \in \textit{phase } 1 \\ \varepsilon^P{}_{ij}(\mathbf{x}) + \varepsilon^v{}_{ij}(\mathbf{x}) & \mathbf{x} \in \textit{phase } 2. \end{cases} \tag{3}$$

The strain energy in the system can then be formulated as the summation of an equivalent strain energy of the homogeneous system which is entirely comprised of the matrix material, and a deviation related to the virtual strains (i.e. differences between the inhomogeneous and homogeneous systems) [31]:

$$E^{strain} = E^{eq} + \Delta E = \frac{1}{2}\int C_{ijkl}{}^{(1)}(\varepsilon_{ij}(\mathbf{x}) - \varepsilon^0{}_{ij}(\mathbf{x}))(\varepsilon_{kl}(\mathbf{x}) - \varepsilon^0{}_{kl}(\mathbf{x}))dV - \frac{1}{2}\int_{phase\ 2}(C_{ijmn}{}^{(1)}\Delta S_{mnpq}(\mathbf{x})C_{pqkl}{}^{(1)} + C_{ijkl}{}^{(1)})\varepsilon_{ij}{}^{v}(\mathbf{x})\varepsilon_{kl}{}^{v}(\mathbf{x})dV, \tag{4}$$

where $\varepsilon_{ij}(\mathbf{x})$ represents the total strain. In the above equation, the first part of integration is taken over all the phases while the second integral is performed only in the second phase since in the case of bi-metal interfaces this is the only region with a non-zero virtual strain.

Material stiffness in phase 1 and phase 2 are defined with superscripts (1) and (2):

$$C_{mnpq}(\mathbf{x}) = \begin{cases} C_{mnpq}{}^{(1)} & \mathbf{x} \in \textit{phase } 1 \\ C_{mnpq}{}^{(2)} & \mathbf{x} \in \textit{phase } 2. \end{cases} \tag{5}$$

Having the above definition, the fourth-order compliance tensor $\Delta S_{mnpq}(\mathbf{x})$, which is used only for the second phase, can be written as follows

$$\Delta S_{mnpq}(\mathbf{x}) = (C_{mnpq}(\mathbf{x}) - C_{mnpq}{}^{(1)})^{-1}. \tag{6}$$

Finally, the total strain for the bi-phase material can be expressed with the following relation

$$\varepsilon_{ij}(x) = \bar{\varepsilon}^0{}_{ij} + \int \widehat{G}_{jk}(\mathbf{k})k_i k_l C^{(1)}{}_{klmn}\hat{\varepsilon}^0{}_{mn}(\mathbf{k})e^{ikx}\frac{d^3k}{(2\pi)^3} + S^{(1)}{}_{ijkl}\sigma_{kl}{}^{appl}, \tag{7}$$

where, $\bar{\varepsilon}^0{}_{ij} = \frac{1}{V}\int \varepsilon^0{}_{ij}(\mathbf{x})\,dV$ is the volume average stress-free strain. Substituting Eqs. (3), (6) and (7) into Eq. (4), the strain energy for a bi-phase material can be written as [19,20]

$$E^{strain} = E^{eq} + \Delta E = \frac{1}{2}\int \widehat{A}_{mnuv}(\mathbf{k})(\hat{\varepsilon}^v{}_{mn}(\mathbf{k}) + \hat{\varepsilon}^p{}_{mn}(\mathbf{k}))(\hat{\varepsilon}^{v*}{}_{uv}(\mathbf{k}) + \hat{\varepsilon}^{p*}{}_{uv}(\mathbf{k}))\frac{d^3k}{(2\pi)^3}$$
$$- \frac{V}{2}S_{ijkl}{}^{(1)}\sigma_{ij}{}^{appl}\sigma_{kl}{}^{appl} - \sigma_{ij}{}^{appl}\int(\varepsilon^v{}_{ij}(\mathbf{x}) + \varepsilon^p{}_{ij}(\mathbf{x}))\,dV$$
$$- \frac{1}{2}\int_{phase\ 2}(C^{(1)}{}_{ijmn}\Delta S_{mnpq}(\mathbf{x})C^{(1)}{}_{pqkl} + C^{(1)}{}_{ijkl})\varepsilon^v{}_{ij}(\mathbf{x})\varepsilon^v{}_{kl}(\mathbf{x})dV, \tag{8}$$

where $\widehat{A}_{mnuv}(\mathbf{k}) = C^{(1)}{}_{mnuv} - C^{(1)}{}_{kluv}C^{(1)}{}_{ijmn}\widehat{G}_{ki}(\mathbf{k})k_j k_i$. The last term in Eq. (8) represents the elastic strain energy resulting from the inhomogeneity (i.e. differences in elastic moduli) of phase 2 relative to phase 1.

In a bi-material, the lattice parameters of each material, $a^{(1)}$ and $a^{(2)}$, are not equal in general and such a difference causes a misfit strain in order to maintain coherency at the interface. For the PFDD formulation to account for the misfit strain, the total strain in Eq. (7) is extended to include the misfit strain. Here we limit our study to cube-on-cube, coherent or semi-coherent (with wide spacing between misfit dislocations) interfaces. For this type of interface, we can assume plane stress in the plane of interface. With these assumptions, the misfit strain in a local interface coordinate system (see Fig. 1 for illustration) for phase 1 and phase 2 can be defined as following [35]

$$\varepsilon_{ij}{}^{mis,loc}(\mathbf{x}) = \begin{pmatrix} \varepsilon_{11}{}^{(1)} & 0 & 0 \\ 0 & \varepsilon_{22}{}^{(1)} & 0 \\ 0 & 0 & \varepsilon_{33}{}^{(1)} \end{pmatrix}\left(1 - \frac{d(\mathbf{x})}{d^{(1)}}\right), \quad \mathbf{x} \in \text{phase } 1$$
$$\varepsilon_{ij}{}^{mis,loc}(\mathbf{x}) = \begin{pmatrix} \varepsilon_{11}{}^{(2)} & 0 & 0 \\ 0 & \varepsilon_{22}{}^{(2)} & 0 \\ 0 & 0 & \varepsilon_{33}{}^{(2)} \end{pmatrix}\left(1 - \frac{d(\mathbf{x})}{d^{(2)}}\right), \quad \mathbf{x} \in \text{phase } 2, \tag{9}$$

where $d(\mathbf{x})$ is the normal distance from the interface to any point in either material, and $d^{(1)}$, $d^{(2)}$ are the cutoff values of $d(\mathbf{x})$in materials 1 and 2, respectively. Fig. 1 illustrates this configuration. The misfit strain components $\varepsilon_{ij}{}^{(1)}$, $\varepsilon_{ij}{}^{(2)}$ can be written as [35–37]

$$\varepsilon_{11}{}^{(1)} = \varepsilon_{22}{}^{(1)} = \frac{C_{eff}{}^{(2)}(a^{(2)} - a^{(1)})}{C_{eff}{}^{(1)}a^{(2)} + C_{eff}{}^{(2)}a^{(1)}}; \quad \varepsilon_{33}{}^{(1)} = \frac{-2\nu^{(1)}\varepsilon_{11}{}^{(1)}}{1 - \nu^{(1)}}$$
$$\varepsilon_{11}{}^{(2)} = \varepsilon_{22}{}^{(2)} = \frac{C_{eff}{}^{(1)}(a^{(1)} - a^{(2)})}{C_{eff}{}^{(1)}a^{(2)} + C_{eff}{}^{(2)}a^{(1)}}; \quad \varepsilon_{33}{}^{(2)} = \frac{-2\nu^{(2)}\varepsilon_{11}{}^{(2)}}{1 - \nu^{(2)}}, \tag{10}$$

where $C_{eff} = \frac{E}{1-\nu}$, and E, ν are the Young's modulus and Poisson's ratio, respectively. Variables ε_{11}, ε_{22} are in-plane normal strain components in x_1, x_2 direction and ε_{33} is the out-of-plane normal strain component perpendicular to the interface plane and in x_3 direction.

In order to incorporate the misfit strain into the PFDD formulation, a transformation from the local interface frame to the global frame must be performed

$$\varepsilon_{ij}{}^{misfit}(\mathbf{x}) = Q_{ik}Q_{jl}\varepsilon_{kl}{}^{misfit,local}(\mathbf{x}), \tag{11}$$

where Q_{ij} denotes the transformation matrix from the local interface

Fig. 1. Bi-phase interface system showing phases, misfit strain region, and the interface coordinate frame.

coordinate system (x_1, x_2, x_3) to the global coordinate system (x, y, z).

Adding the misfit strain into the total strain in Eq. (7), we have [20]

$$\varepsilon_{ij}(\mathbf{x}) = \bar{\varepsilon}^0{}_{ij} + \int \widehat{G}_{jk}(\mathbf{k}) k_i k_l C^{(1)}{}_{klmn} \hat{\varepsilon}^0{}_{mn}(\mathbf{k}) e^{ikx} \frac{d^3k}{(2\pi)^3} + S^{(1)}{}_{ijkl}\sigma_{kl}{}^{appl} + \varepsilon_{ij}{}^{misfit}(\mathbf{x}), \tag{12}$$

Then by incorporation of Eq. (12) into Eq. (4), the new form of the strain energy is derived for a bi-phase material for which the interface misfit strain is accounted [20]

$$\begin{aligned} E^{strain} = E^{eq} + \Delta E = {} & \frac{1}{2}\int \widehat{A}_{mnuv}(\mathbf{k})(\hat{\varepsilon}^v{}_{mn}(\mathbf{k}) + \hat{\varepsilon}^p{}_{mn}(\mathbf{k}))(\hat{\varepsilon}^{v*}{}_{uv}(\mathbf{k}) \\ & + \hat{\varepsilon}^{p*}{}_{uv}(\mathbf{k}))\frac{d^3k}{(2\pi)^3} \\ & - \frac{1}{2}\int C^{(1)}{}_{ijkl}\varepsilon_{ij}{}^{misfit}(\mathbf{x})\varepsilon_{kl}{}^{misfit}(\mathbf{x})dV \\ & - C^{(1)}{}_{ijkl}\int (\varepsilon^v{}_{ij}(\mathbf{x}) + \varepsilon^p{}_{ij}(\mathbf{x}))\varepsilon_{kl}{}^{misfit}(\mathbf{x})dV \\ & - \frac{V}{2}S_{ijkl}{}^{(1)}\sigma_{ij}{}^{appl}\sigma_{kl}{}^{appl} - \sigma_{ij}{}^{appl}\int (\varepsilon^v{}_{ij}(\mathbf{x}) + \varepsilon^p{}_{ij}(\mathbf{x}))dV \\ & + \int S^{(1)}{}_{ijkl}\sigma_{ij}{}^{appl}\varepsilon_{kl}{}^{misfit}(\mathbf{x})dV \\ & - \frac{1}{2}\int_{phase\ 2}(C^{(1)}{}_{ijmn}\Delta S_{mnpq}(\mathbf{x})C^{(1)}{}_{pqkl} + C^{(1)}{}_{ijkl})\varepsilon^v{}_{ij}(\mathbf{x})\varepsilon^v{}_{kl}(\mathbf{x})dV. \end{aligned} \tag{13}$$

Additional terms in Eq. (13) comparing to Eq. (8) are the misfit strain energy terms.

2.2. Energy to form a residual dislocation in the interface

As described in the previous section, the lattice parameters in the two materials are generally not equal. Hence, the Burgers vectors for gliding dislocations are also not equal. When a dislocation passes from material 1 to material 2, the Burgers vectors must be conserved in order to maintain conservation of mass. Because the Burgers vectors are not equal in the two materials, a residual Burgers vector is left in the interface defined as $\mathbf{b}^r = \mathbf{b}^{(2)} - \mathbf{b}^{(1)}$ after the dislocation has passed from material 1 to material 2. The energy required to form this residual Burgers vector in the interface region is included as its own term in the total system energy, as shown in Eq. (2) as E^{res}.

In PFDD, since we are only considering cube-on-cube interfaces, there is only one active order parameter required to model the transmission of a single dislocation through the bi-phase interface. This is because the slip direction and slip plane normal does not change across the interface for cube-on-cube alignment. Furthermore, after the dislocation has transmitted through the interface, the order parameter will equal one in both materials indicating that a dislocation has traversed both materials passing through the interface. Within the interface region, defined by two bounding planes: one contributed from material 1 and one contributed from material 2, the order parameter will be non-zero and non-integer to account for the residual dislocation, which is not a full dislocation belonging to a slip system in either material 1 or material 2. In order to calculate the displacement across the interface due to the presence of the residual dislocation, we introduce the notation where $\mathbf{x}^{(I1)}$ are the subset of points that make up the interface plane contributed by material 1, and similarly $\mathbf{x}^{(I2)}$ are the subset of points that make up the interface plane contributed by material 2. Then the displacement across the interface due to the residual dislocation can be defined as:

$$\mathbf{u}_r = \xi^{(2)}\mathbf{b}^{(2)} - \xi^{(1)}\mathbf{b}^{(1)}, \tag{14}$$

where $\xi^{(1)}$ and $\xi^{(2)}$ is shortened notation for $\xi(\mathbf{x}^{(I1)})$ and $\xi(\mathbf{x}^{(I2)})$, respectively. The displacement defined in Eq. (14) will induce tractions in both materials near the interface. In this region the energy required for a residual dislocation can be defined as [20]:

$$E^{res} = \int_S |\boldsymbol{\tau}^{(2)}.\mathbf{u}_r{}^{(2)} - \boldsymbol{\tau}^{(1)}.\mathbf{u}_r{}^{(1)}|\, dS, \tag{15}$$

where $\boldsymbol{\tau}^{(1)}$ and $\boldsymbol{\tau}^{(2)}$ are the tractions in materials 1 and 2, respectively. These tractions can be calculated using the stiffness tensors in each material, the slip direction, slip plane normal, and order parameter values. Details can be found in [20].

2.3. Core energy

The core energy in Eq. (2) accounts for the energy required to move a dislocation line (or core) through the crystal lattice by breaking and reforming atomic bonds. For perfect dislocations, this can be modeled using a Fourier sine series since its sinusoidal nature mimics the periodic nature of the regular atomic lattice in a cubic material [19,38,39]:

$$E^{core}(\xi) = \sum_{\alpha=1}^{N}\int B\sin^2(n\pi\,\xi_\alpha(\mathbf{x},\,t)\,)\delta_n dV, \tag{16}$$

where B defines the energy barrier magnitude which is overcome to activate the slip. B is calculated using the model from [40] and is dependent on material parameters such as the stiffness tensor, slip plane normal, and inter-planar distance between slip planes.

We highlight that due to the physical understanding we have of this energy term, there is no dependence on the virtual strain terms. In addition, we note that this formulation and the subsequent study are restricted to perfect dislocation motion in the interest of simplicity. Partial dislocation behavior has been included in the PFDD formulation, and depends upon energy surfaces calculated using atomistic methods. More details on this formulation can be found in [19,41].

2.4. Minimization using the Ginzburg–Landau equation

In the PFDD formulation, the system evolution is carried out through the minimization of total energy. The equations used for such minimizations are the time-dependent Ginzburg–Landau (TDGL) kinetic equations, in which the phase field variable is related to the total system energy as [32,38,42]

$$\frac{\partial\xi_\alpha(\mathbf{x},\,t)}{\partial t} = -L\frac{\delta E(\xi)}{\delta\xi_\alpha(\mathbf{x},\,t)}, \tag{17}$$

where L is a kinetic coefficient controlling the simulation time scale. Eq. (17) presents a set of coupled integro-differential equations, which describe the dislocation dynamics of the entire system. The location of each and every dislocation is determined by solving this set of equations through the entire system evolution. The virtual strain components evolve in a similar manner using the following equation:

$$\frac{\partial\varepsilon_{ij}{}^v(\mathbf{x},\,t)}{\partial t} = -K\frac{\delta E(\xi,\,\boldsymbol{\varepsilon}^v)}{\delta\varepsilon_{ij}{}^v(\mathbf{x},\,t)},\ \text{in phase 2} \tag{18}$$

where K is a material constant related to heterogeneity of phase 2.

3. Validation and application case studies

3.1. Validation benchmark: stress field around an edge dislocation

In order to validate the newly developed ACCPFDD code, a case study has been performed, in which the stress field around an infinitely long edge dislocation has been calculated and compared to the analytical solution. The following equations define the stress field equations around an infinitely long edge dislocation [43]

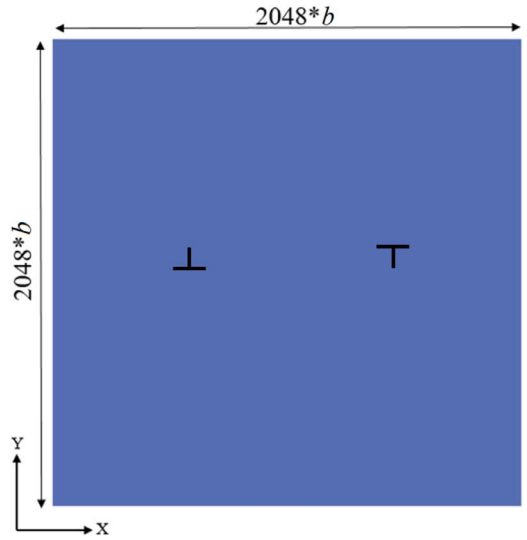

Fig. 2. Initial configuration of the simulation setup to validate ACCPFDD implementation by comparison of the stress field around an infinite edge dislocation. The positive and negative dislocation symbols show the actual location of the positive and negative dislocations comprising a dislocation dipole.

Table 1
Copper parameters for study of stress field around an infinite edge dislocation [20].

Material constant	Value
C_{11}(GPa)	168.4
C_{12}(GPa)	121.4
C_{44}(GPa)	23.5
μ (GPa)	23.5
b (nm)	0.256
a (nm)	0.361
ν	0.41891

$$\sigma_{xx} = -\frac{\mu b}{2\pi(1-\nu)}\frac{y(3x^2+y^2)}{(x^2+y^2)^2}$$
$$\sigma_{yy} = \frac{\mu b}{2\pi(1-\nu)}\frac{y(x^2-y^2)}{(x^2+y^2)^2}$$
$$\sigma_{xy} = \frac{\mu b}{2\pi(1-\nu)}\frac{x(x^2-y^2)}{(x^2+y^2)^2} \quad (19)$$

where $\boldsymbol{\sigma}$, μ, b, and ν represent the stress tensor, shear modulus, value of the Burgers vector, and Poisson's ratio, respectively. Fig. 2 depicts the initial configuration for the corresponding simulation in the ACCPFDD code. We set the material in both phases to be copper with the material parameters provided in Table 1 [20]. In order to satisfy periodic boundary conditions, a dislocation dipole has been used for the initial conditions in the ACCPFDD simulation so that the net Burgers vector in the cell is zero. Indeed, the plus and minus signs in Fig. 2 represent the actual locations of the initial dislocations with opposite signs. Fig. 3 depicts the comparison of contours between the ACCPFDD simulation and the analytical solution. Qualitatively, contour shapes are in good agreement; however, in order to make the comparison quantitative, we discretize the analytical solution data points in MATLAB and then post process the output in PARAVIEW to make a consistent comparison with ACCPFDD stress field output. Figs. 4, 5, and 6 represent the stress variation along a horizontal line extended from one side to the other side of the cell and positioned 10b and 50b above the centerline.

Quantitative results show good agreement between ACCPFDD and the analytical solution. It is notable that, at the origin (i.e., dislocation line position), due to the singularity in the stress field in Eq. (19), perturbations and fluctuations are observed in ACCPFDD code that are unavoidable due to the numerical nature of the solver. Moreover, with careful examination of Figs. 6i–6iii, we see that far away from the dislocation position the ACCPFDD and analytical solution trends are the same but the curves are not on top of each other. This is more evident when the extracted data line is farther from the centerline of the dislocation (e.g., 50b comparing to 10b). These differences may be present for two main reasons. First, there can be an effect from the stress fields of the dislocations in the surrounding periodic cells. Such an interaction could potentially cause large variations in the stress field if the simulation cell is small enough. We have attempted to mitigate this effect by choosing a large simulation cell size. For these simulations, the simulation cell size is set to $2048b \times 2b \times 2048b$, where the dislocations are placed 1024b far apart. This results in very large arrays with a size of approximately 55 billion elements. So far, due to the memory limitations (both CPU and GPU), we are not able to go beyond this size. We also note that due to the periodic boundary conditions, stress fields from neighboring dislocations will always be present necessitating large cell sizes that will be much easier to achieve with the enhanced computational efficiency of the ACCPFDD code.

The second and more likely cause for the deviations present in Figs. 6i–6iii is the interaction between the stress fields of a dislocation dipole that is present in the ACCPFDD simulation but not the analytical solution, which treats a single isolated dislocation. Hence, in the ACCPFDD simulation the stress field surrounding the dislocation monopoles will be slightly perturbed due to elastic interactions between the two dislocations. As we move on a horizontal line away from the dislocation core region, these perturbations become more visible since the stress fields are lower in magnitude (see y-axes in Fig. 6). In order to resolve this issue completely, a larger domain may help because the two dislocation monopoles will be farther apart and therefore interact less. However, since we are comparing the stress field around one dislocation that is a part of a dipole in our code with a single dislocation in the analytical solution, regardless of the domain size there will some small differences due to the presence of the additional dislocation in the

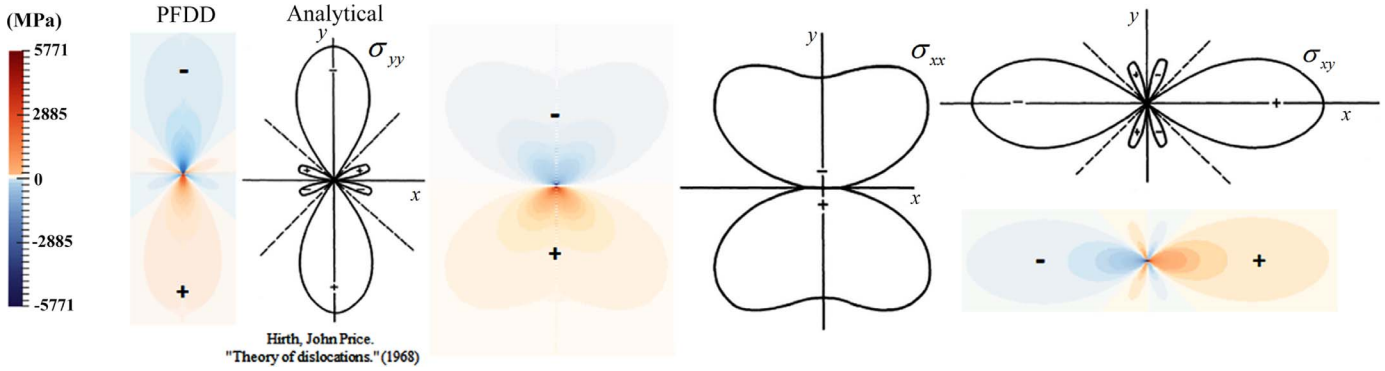

Fig. 3. Cauchy stress field around an infinite edge dislocation: ACCPFDD contours vs. analytical solution [43]. The plus and minus signs indicate the sign of the stress orbitals.

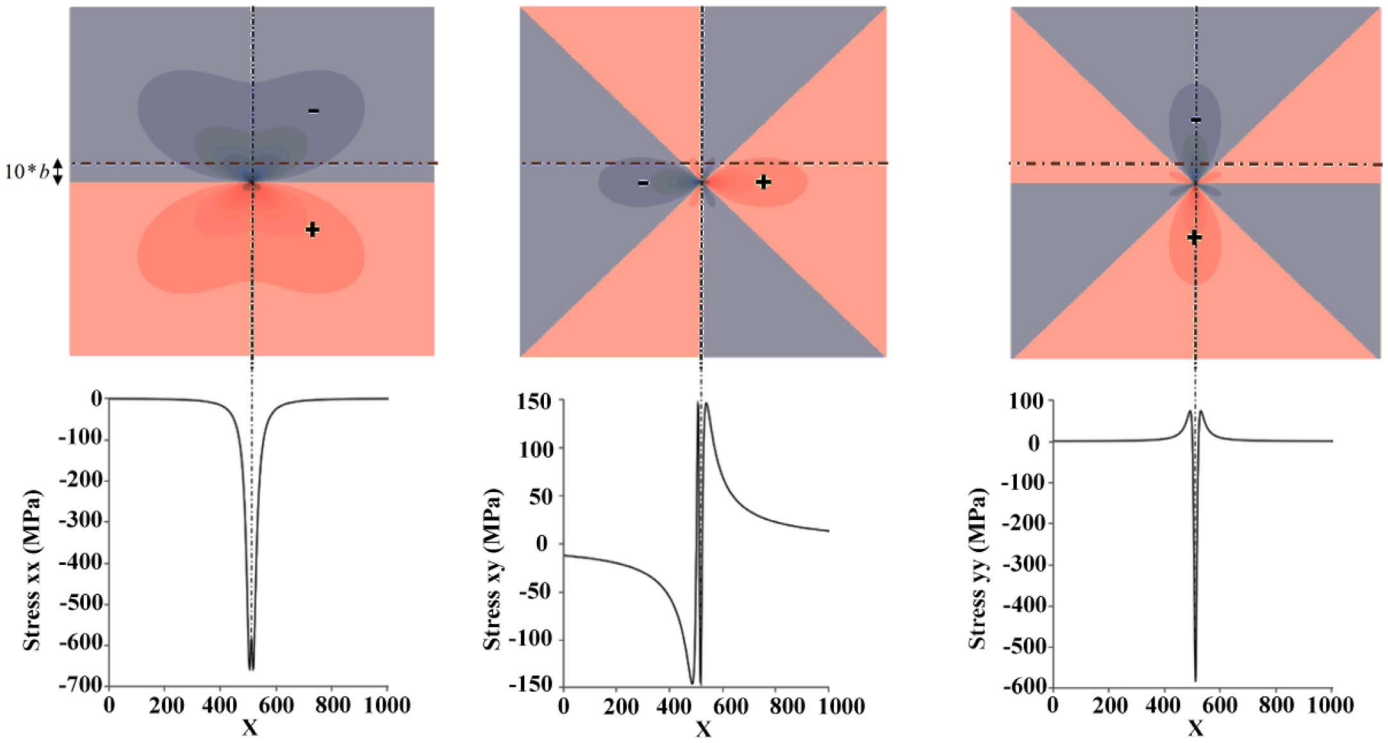

Fig. 4. Stress field around an infinite edge dislocation based on the analytical solution.

numerical simulations. Nevertheless, the ACCPFDD and the analytical solutions while not in exact agreement, are still in very good agreement.

In order to ensure the comparisons made here are accurate and consistent, we take a further step and compare the ACCPFDD data with the analytical solution of stress fields around an infinite array of edge dislocations with periodic boundary conditions provided by Hirth [43] and Srolovitz and Lomdahl [44]. The analytical solution of stress field around an infinite array of dislocations spaced by a distance D from each other can be written as follows:

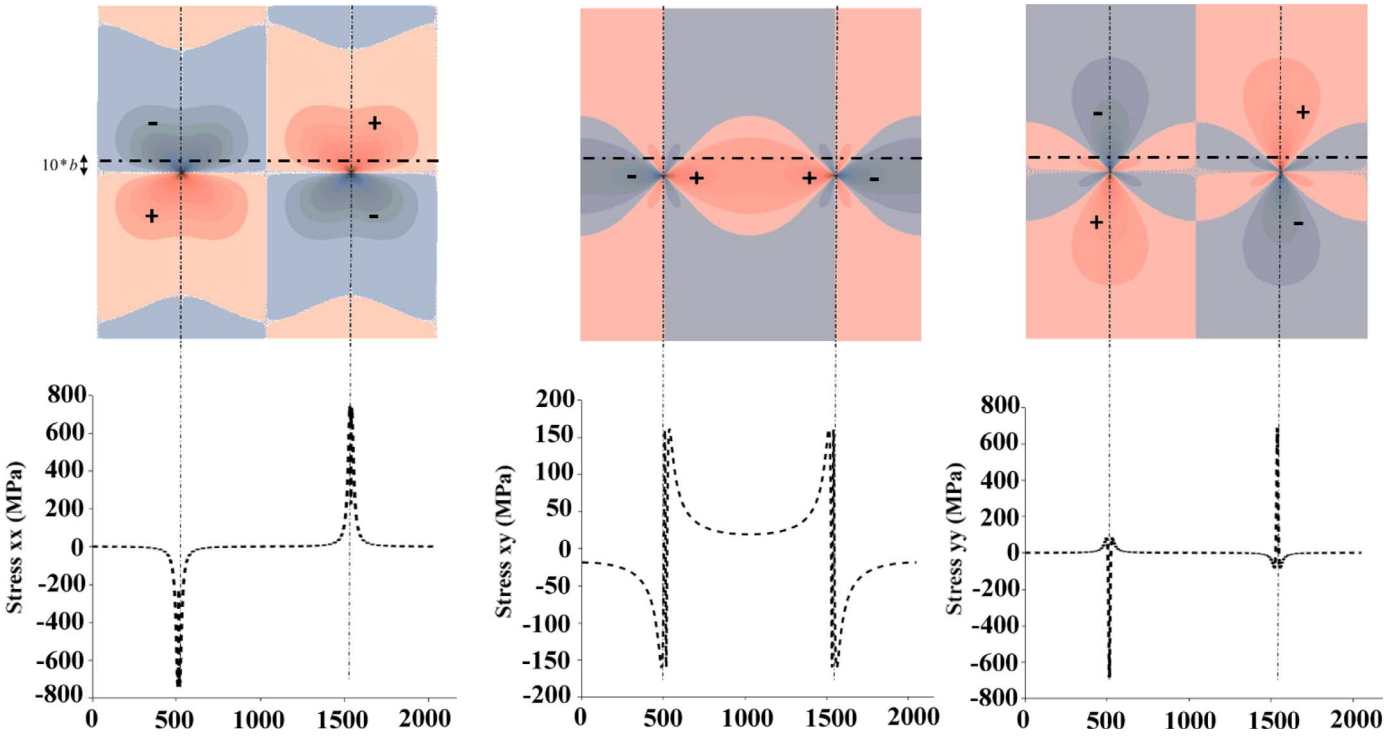

Fig. 5. Stress field around an infinite edge dislocation calculated using ACCPFDD.

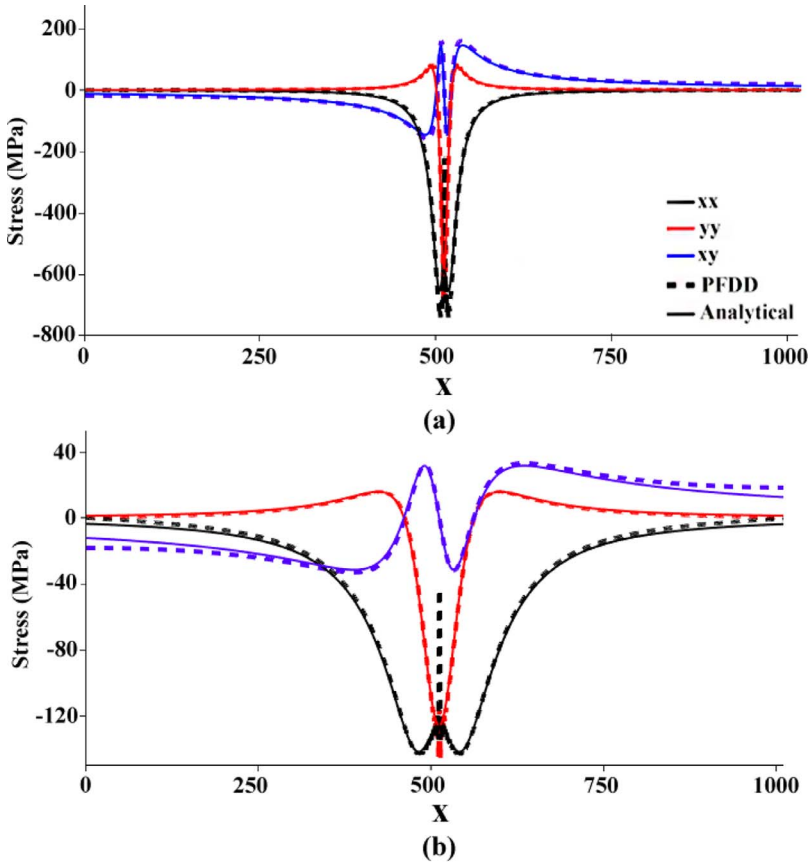

Fig. 6i. Comparison of stress field around an infinite edge dislocation calculated using ACCPFDD and analytically for the domain size of $2048b \times 2b \times 2048b$: (a) $10b$ above the center line and (b) $50b$ above the center line, where b is the magnitude of Burgers vector.

$$\sigma_{xx} = -\sigma_0 2\pi X(\cosh 2\pi X \cos 2\pi Y - 1)$$

$$\sigma_{yy} = -\sigma_0[2\sinh 2\pi X(\cosh 2\pi X - \cos 2\pi Y) - 2\pi X(\cosh 2\pi X \cos 2\pi Y - 1)]$$

$$\sigma_{xy} = \sigma_0 \sin 2\pi Y(\cosh 2\pi X - \cos 2\pi Y - 2\pi X \sinh 2\pi X)$$

$$\sigma_0 = \frac{\mu b}{2D(1-\nu)(\cosh 2\pi X - \cos 2\pi Y)^2} \quad (20)$$

where, X and Y represent the dimensionless coordinates normalized by the spacing D. Fig. 7 represents the stress field and related quantification curves around two edge dislocations placed in an infinitely periodic domain with a size of $2048b \times 2b \times 2048b$ representing the same arrangement of dipoles we presented in Fig. 2. Good agreement is observed between the ACCPFDD and analytical results for the infinite periodic edge dislocation setup as well.

3.2. Application benchmark: a bi-phase Cu–Ni interface case study

After validation of the ACCPFDD code with the analytical results, we performed a case study in which a bi-phase copper-nickel (Cu–Ni) material interface was investigated using the three PFDD versions: ACCPFDD, multicore PFDD, and the original serial PFDD. We emphasize that sole PFDD notation will always refer to the serial version of the code with FOURN FFT subroutine from Numerical Recipes [45]. Fig. 8 illustrates the initial problem setup. The material properties for copper were provided previously in Table 1. The material properties for nickel are provided in Table 2.

In this case study, the bi-phase material is given an applied stress of 0.4 GPa along $[11\bar{2}]$ direction. The misfit strains are set according to the Eqs. (9)–(10). The problem was then simulated with the serial, multicore and ACCPFDD version of the code. Figs. 9 and 10 compare the stress and strain field in the Cu–Ni bi-crystal interface system, respectively, for different problems sizes of $16b \times 2b \times 16b$, $32b \times 2b \times 32b$, $64b \times 2b \times 64b$, and $128b \times 2b \times 128b$ after reaching a minimized state of total system energy through solving the TDGL set of Eq. (17). Results confirm the identical response for different versions of the code (i.e., serial, multicore parallel on 22 CPU cores, and ACCPFDD). For the remainder of this study, this bi-metal benchmark problem will be used in all performance comparison simulations.

4. PFDD GPU implementation

4.1. Profiling the serial PFDD algorithm

In order to accelerate any code whether on CPU or GPU, the first step to take is to determine the hotspots (i.e., most time consuming routines) within the algorithm. In order to profile the PFDD code, the PGI performance profiler (PGPROF) 2016 v16.10 was used [46]. Fig. 11 illustrates the percentage of computational time involved in PFDD solver for different problem sizes (i.e. number of FFT voxels). We found that the energy calculation routine defined in Eq. (13) takes more than 90% of the simulation runtime. More specifically, the calculation of the first term in Eq. (13) encompasses most of the computation time. This term accounts for elastic interaction including dislocation-dislocation interactions (such as attraction and repulsion) and interactions between the plastic and virtual strains. This term also includes the interaction matrix ($\widehat{A}_{mnuv}$ defined following Eq. (8)). The interaction matrix scales not only with the size of the simulation cell, but also with the number of active slip systems for both the plastic component (up to 12 for fcc metals) and virtual component (9 active components) of the strain in Eq. (13). The interaction matrix is the largest calculation done in the algorithm, hence it is not unexpected to find that it dominates the computation time. With respect to other terms in Eq. (13), we note that the second, third, and sixth terms are calculating effects due to the presence of misfit strains at the interface. As can be seen in Fig. 1, the

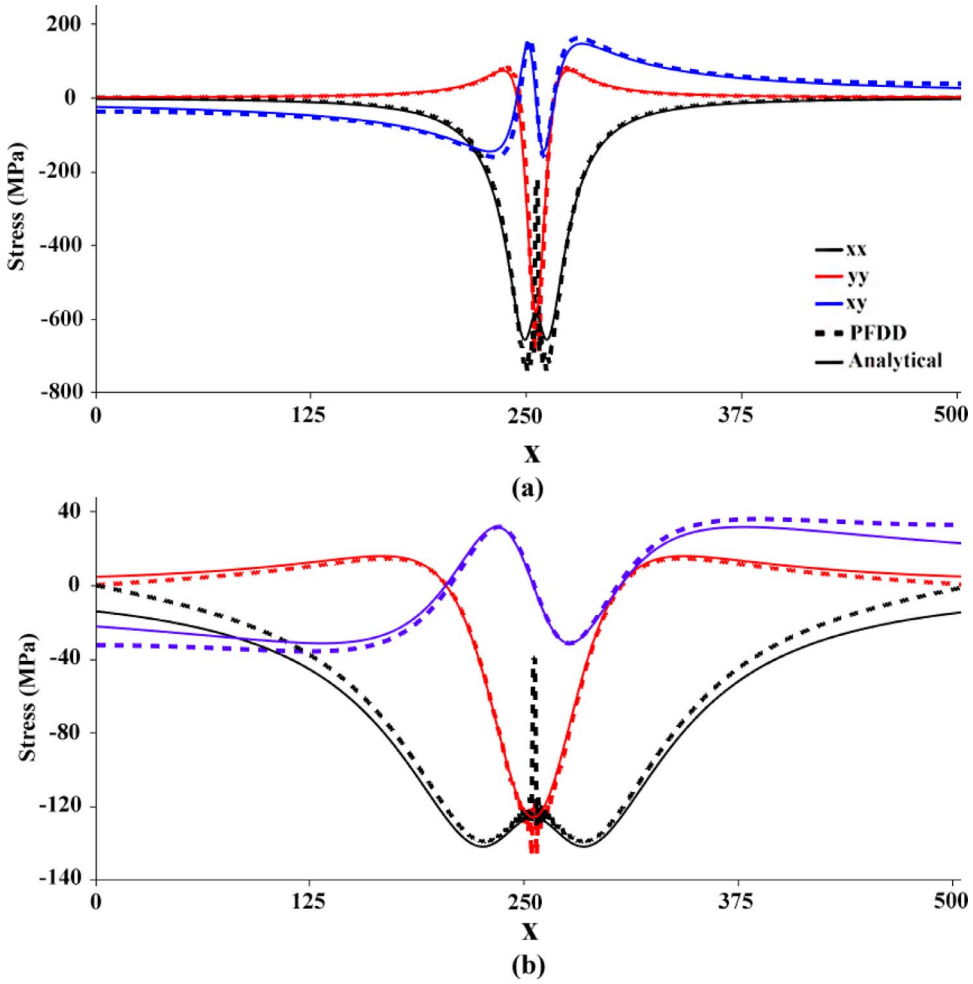

Fig. 6ii. Comparison of stress field around an infinite edge dislocation calculated using ACCPFDD and analytically for the domain size of $1024b \times 2b \times 1024b$: (a) $10b$ above the center line and (b) $50b$ above the center line, where b is the magnitude of Burgers vector.

misfit strains are only non-zero in a localized region surrounding the interface.

Fig. 11 also shows that with increasing domain size, the percentage of computational time required for calculation of the first term in Eq. 13 decreases from 94% to 92% and 90% for problem sizes of 2048, 8192, and 32,768 FFT points, respectively. For convenience, we will refer to this term as the $\widehat{\mathrm{A}} - term$. This implies that there are some other routines, which become more computationally dominant while increasing the problem size. We will elaborate on this point later in the text. Nevertheless, it is clear from the profiling that this subroutine is a hotspot within the PFDD algorithm. We next determine if it can be addressed by porting to GPUs.

The ″-Minfo = ccff ″ flag was enabled to take the advantage of common compiler feedback format (CCFF). When CCFF is enabled, the compiler is fortified with more detailed information output about the optimization and performance diagnostics. Fig. 12 shows a schematic of detailed PGI linker together with PGI performance profiler while the CCFF flag is enabled.

One of the advantages of common compiler feedback is the loop intensity information it provides. Computational intensity defines the ratio of computation (i.e. execution on GPU) to the data movement (i.e. data transfer between CPU and GPU). If the loop intensity magnitude is less than 1, porting the loop to the GPU is not efficient due to the large amount of time being spent on data movement rather than computation itself. On the other hand, a loop intensity greater than 4.0 is favorable for GPU parallelism. Appendix A presents the loop intensity information provided with CCFF information enabled in PGPROF for the nested loops within the subroutine identified as a hotspot (i.e. where the first term in Eq. (13) is calculated). The three outermost loops for the calculation of the interaction matrix, $\widehat{A}_{mnuv}$ ($\widehat{\mathrm{A}}$), possess intensities of 385.34, 380.49, and 375.61, respectively, for the problem size of 2048 FFT voxels. Additionally, the innermost loop intensity is 4.00, which satisfies the minimum requirement of loop intensity (i.e. loop intensity > 1.0) for efficient GPU implementation. A thorough evaluation of the interaction matrix calculation indicates that it consists of 11 nested loops of which the 3 outermost are tightly nested. This loop structure makes it a potential candidate for GPU parallelism, because of the highly floating point intense operations performed within those three outermost loops.

4.2. OPENACC implementation of PFDD – ACCPFDD

OPENACC, originally developed by three major vendors CAPS [47], CRAY [48], and PGI [49], is a high level programming model based on directives that are added to annotate the code. The main reason behind using OPENACC is to maintain performance portability of a given code. With only few modifications applied to the existing code, added OPENACC directives result in a high performance GPU code, which can be compiled for various architectures including both GPUs and multicore CPUs. Schematically, Fig. 13 illustrates how both the serial blocks of a code, and the OPENACC parallel loops are executed on CPU and GPU, respectively. Starting with OPENACC 2.0, one is able to run an OPENACC code on the GPU and multicore CPUs (i.e. similar to OPENMP standard for CPU parallelism) at the same time without additional modification to the code. To this end, ″-ta = tesla″ flag is changed to ″-ta = multicore″ to change the PGI compiler target from GPU to multicore CPUs (i.e. multi-threaded application). Moreover, it has been seen that in some cases, OPENACC can result in a better

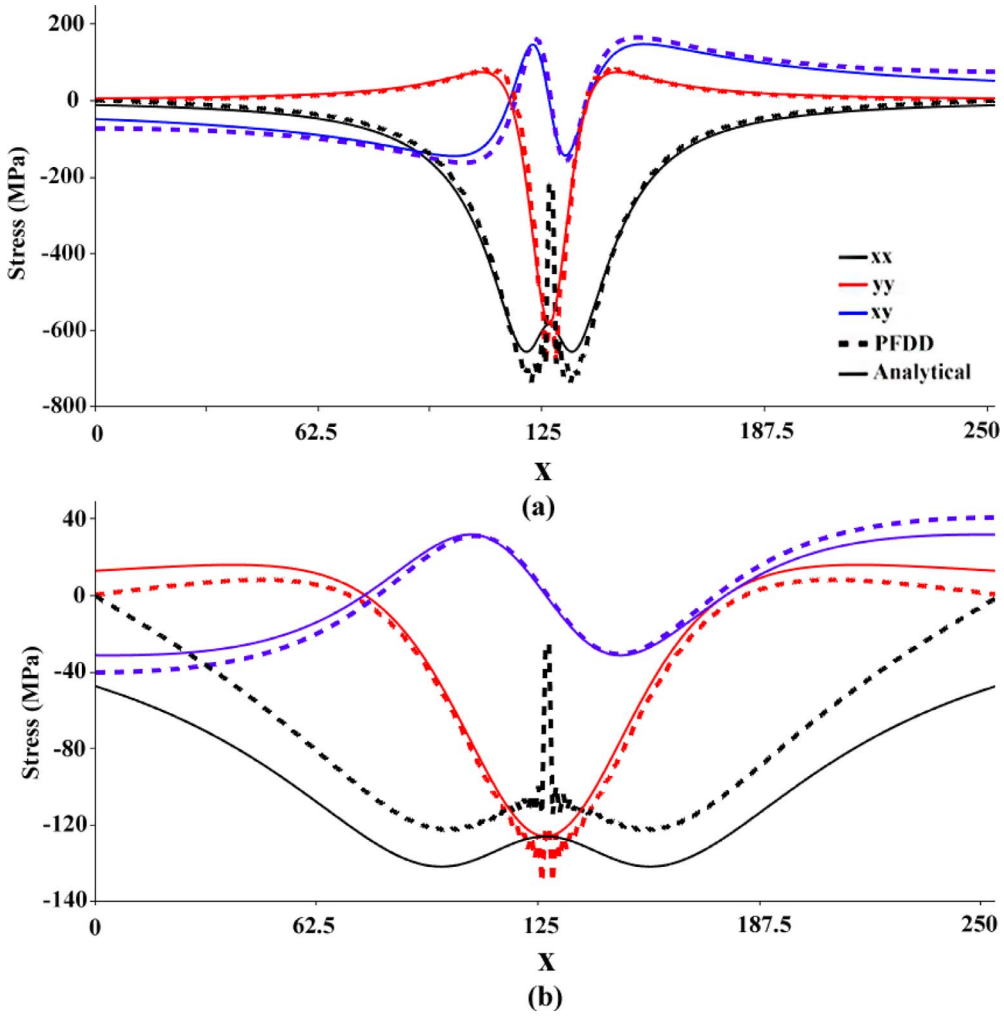

Fig. 6iii. Comparison of stress field around an infinite edge dislocation calculated using ACCPFDD and analytically for the domain size of $512b \times 2b \times 512b$: (a) $10b$ above the center line and (b) $50b$ above the center line, where b is the magnitude of Burgers vector.

efficiency comparing to its peers CUDA and OPENCL [50,51] due to the high level of available optimizations provided with OPENACC. For instance, the flag ″-ta = tesla:fastmath″ which was used in compiling our code, allows the PGI compiler to take advantage of hardware accelerated math routines with an emphasis on performance over accuracy in number of floating points, which under proper circumstances results in a better efficiency while maintaining accuracy.

Given the above information, the subroutine calculating the first term in Eq. (13) including the interaction matrix was ported to GPU using OPENACC directive pragmas. Appendix B describes how we ported the loops inside this subroutine using OPENACC kernel and data pragmas. Readers are encouraged to refer to [51] for detailed information about OPENACC and how to implement it efficiently for porting a code to run on GPU.

In order to keep the data resident on the GPU as much as possible, more loops were ported to run on the GPU using OPENACC. This allowed us to minimize the data transfer specifically for temporary arrays calculated at one loop and used for the subsequent loop. At each and every step of porting loops to GPU, the whole code was profiled carefully to see how removing the data transfer incrementally affects the resulting efficiency. Moreover, we used the OPENACC ″collapse″ feature to unite tightly nested loops. This increased the parallelism for the GPU. Once nested loops are united (i.e., combined together), a larger amount of parallelism is exposed to thousands of GPU threads. The NVIDIA GPU architecture is built on multithreaded streaming multiprocessors (SM). When an OPENACC or CUDA program runs, grid blocks are distributed to multiprocessors. Threads of a thread block execute concurrently on the multiprocessor. A multiprocessor is designed to execute hundreds of threads concurrently. A GPU circuit consists of grids of blocks of threads. Threads are executed in groups of 32 parallel threads that are called warps. The total number of threads and blocks that are used to run a loop on a GPU affect the performance efficiency depending on the size and structure of the ported loop. NVIDIA GPUs exploit a unique architecture called SIMT (single instruction multiple thread) to manage large amount of threads running in groups of warps. In order to tune the loop level parallelism using OPENACC, the ″gang″ and ″vector″ features are used to control the total number of blocks and threads per block used. It is also worth mentioning that ″gang″ and ″vector″ directive clauses in OPENACC resemble the ″blockIdx″ and ″threadIdx″ features available in CUDA, respectively.

Fig. 14 represents the speed up of the major part of the energy calculation routine (the $\widehat{A} - term$) after running parallel on one TESLA K80 GPU for problem sizes of 2048, 8192, and 32,768, respectively. Please also note that when the problem size is mentioned here, cell size is the quantity of interest. The domain is three dimensional having two layers in thickness (i.e. $N \times 2 \times N$; where $N = 32, 64, 128$, etc.); however, the real data size and loop counts in the hotspot subroutine is much bigger than the physical cell size (i.e. $N \times 2 \times N \times 81 \times 81$; where $N = 32, 64, 128,\ldots$) because of the 11 nested loops within the subroutine. For instance, in the case of a cell size of 512 FFT voxels, “3,439,853,568” total loop iterations times the operations included (i.e. total number of additions, subtractions, multiplications and divisions) results in the total number of loop floating point operations (FLOPS) completed within the subroutine. Fig. 14 shows that the calculation of the first term in Eq. (13) is up to 56 times faster in the ACCPFDD code,

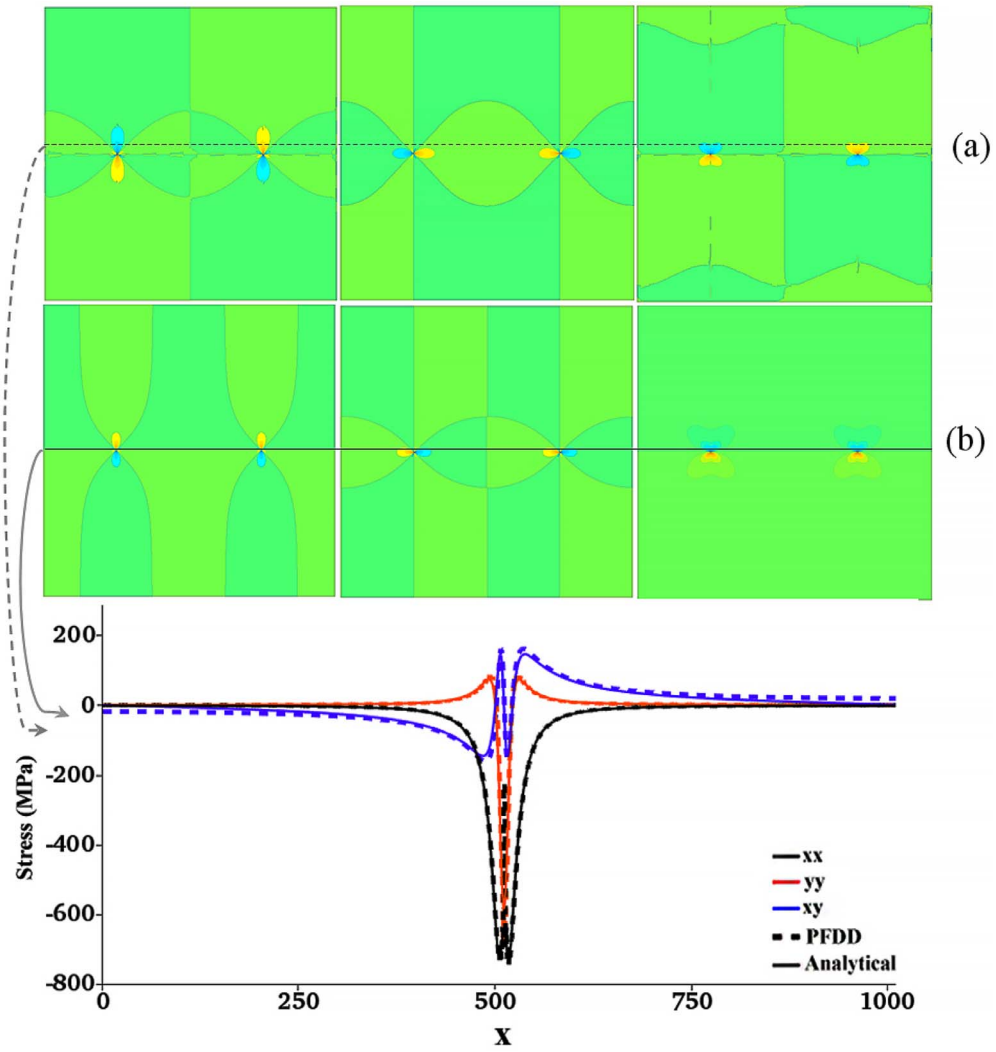

Fig. 7. Comparison of stress field around periodic infinite edge dislocation calculated using ACCPFDD and analytically from Eq. (20) for the domain size of $2048b \times 2b \times 2048b$ on $10b$ above the center line (a) ACCPFDD (b) Analytical.

in comparison to the serial version running on single Intel Xeon CPU. Also, after a problem size of 8192 (i.e.64 × 2 × 64), speedup does not increase any further.

As it was noted previously in the text, the OPENACC code can be compiled with the ″-ta = multicore″ flag to run on multicore CPUs. Fig. 15 represents the PFDD speedup when the hotspot subroutine is run

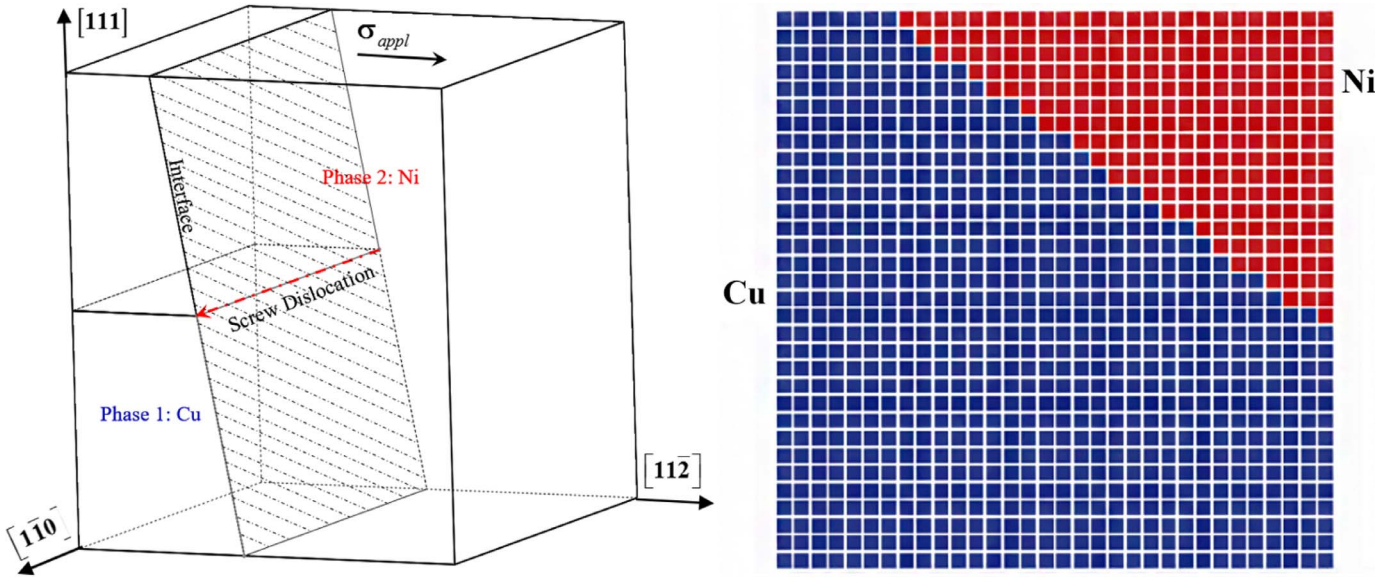

Fig. 8. Initial copper-nickel bi-phase material interface system under applied stress for screw dislocation.

Table 2
Nickel parameters for study of stress field around an infinite edge dislocation [20].

Material constant	Value
C_{11}(GPa)	246.5
C_{12}(GPa)	147.3
C_{44}(GPa)	49.6
μ (GPa)	49.6
b (nm)	0.249
a (nm)	0.352
ν	0.374

on 22 Intel Xeon CPU E5-2699 v4 cores. After 16 CPUs, no more speedup is obtained which is due to the memory bound behavior while running a shared-memory-threaded program with OPENMP or OPENACC multicore parallelism.

After the comparison for the hotspot is done, we measure the total speedup for the whole code. In order to make a comparison between GPU and multicore CPU performance, we compared the total speedup of the PFDD code running the hotspot subroutine on one GPU and the 22-core CPU in Fig. 16. Running the hotspot subroutine on a single Tesla GPU is up to 4 times faster than running it on the 22 cores of the Intel Xeon CPU clocked at 2.20 GHz (scaling up to 3.60 GHz when turbo-boost enabled). The superiority of GPU over CPU computation is appreciated at this point.

It is widely accepted that increasing the problems size should result in an improvement in efficiency for simulations run in parallel on GPUs. The fact that the GPU architecture is designed for massive parallelism (i.e., GPUs are designed to include thousands of built-in CUDA cores) justifies the statement. While we managed to obtain up to 12 times faster

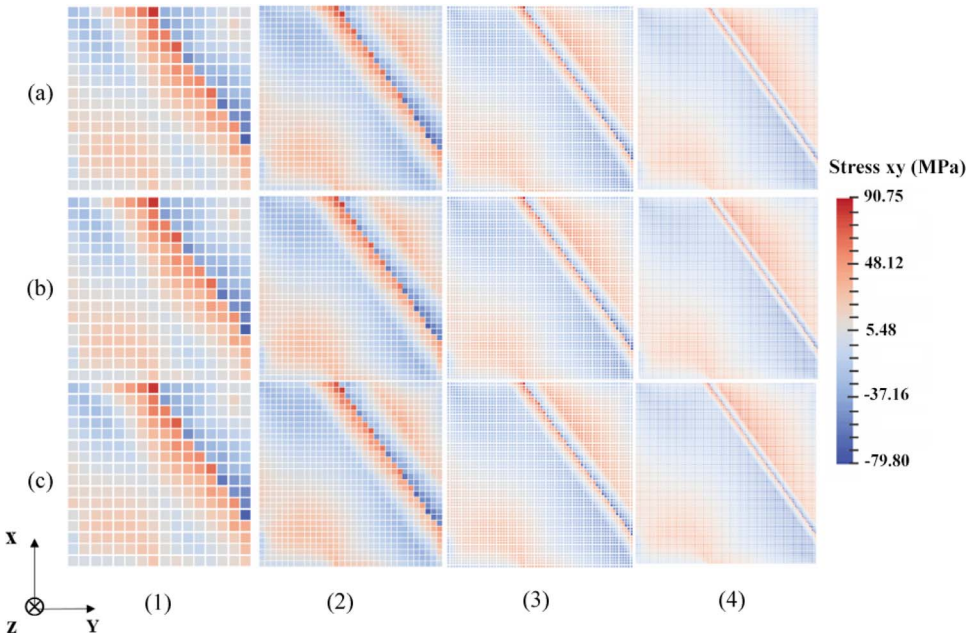

Fig. 9. Stress field around the interface region of a Cu–Ni interface: (a) serial PFDD (b) multicore PFDD (c) ACCPFDD. The results are presented as a function of resolution: (1) 16 × 2 × 16, (2) 32 × 2 × 32, (3) 64 × 2 × 64, and (4) 128 × 2 × 128 FFT voxels.

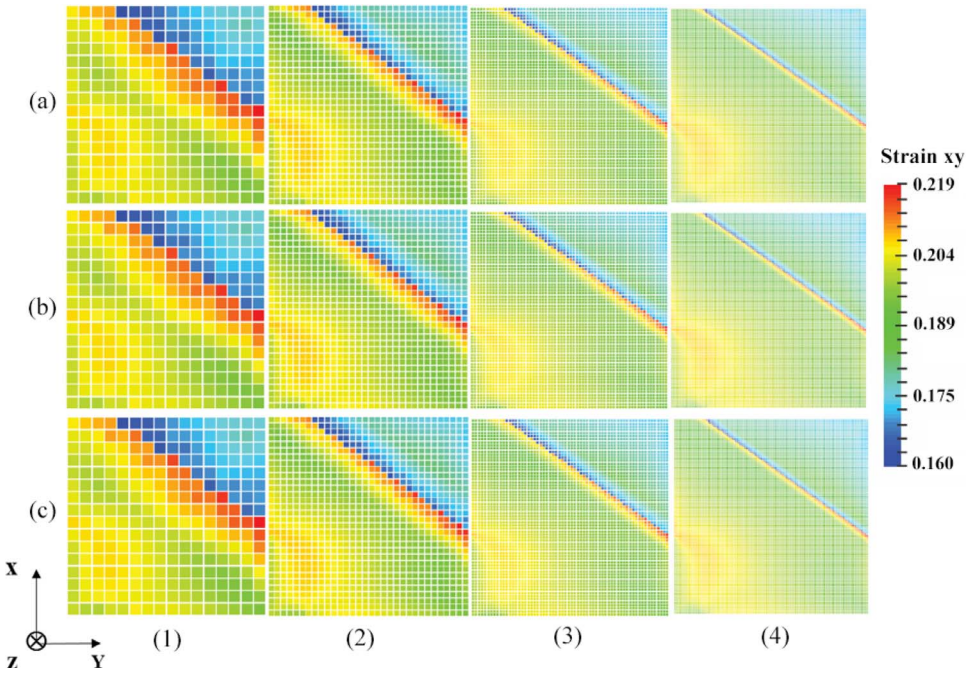

Fig. 10. Strain field around the interface region of a Cu–Ni interface: (a) serial PFDD (b) multicore PFDD (c) ACCPFDD. The results are presented as a function of resolution (1) 16 × 2 × 16, (2) 32 × 2 × 32, (3) 64 × 2 × 64, and (4) 128 × 2 × 128 FFT voxels.

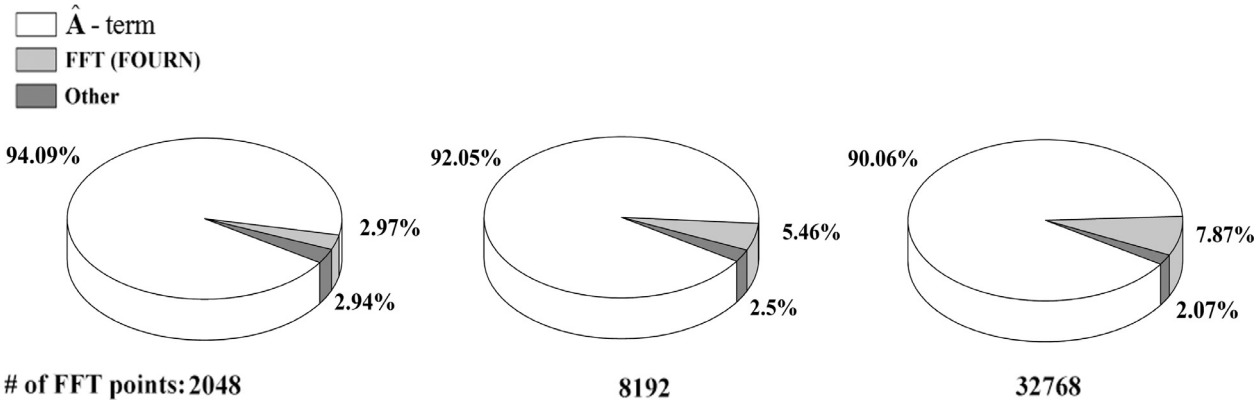

Fig. 11. Percentage of computational time involved in PFDD solver as a function of problem sizes. The $\hat{A} - term$ from the energy Eq. (13) takes more than 90% of the code.

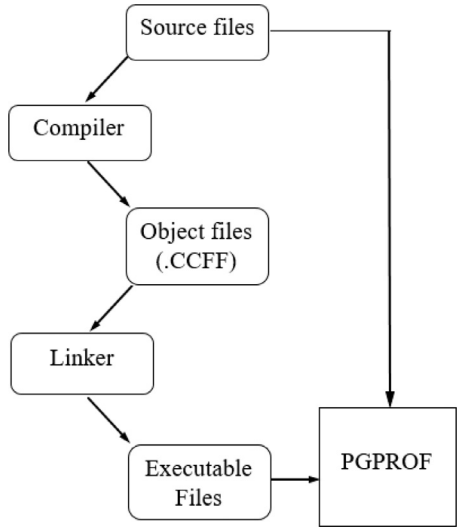

Fig. 12. A schematic of PGI compiler and performance profiler linking flow when common compiler feedback format (CCFF) flag is enabled.

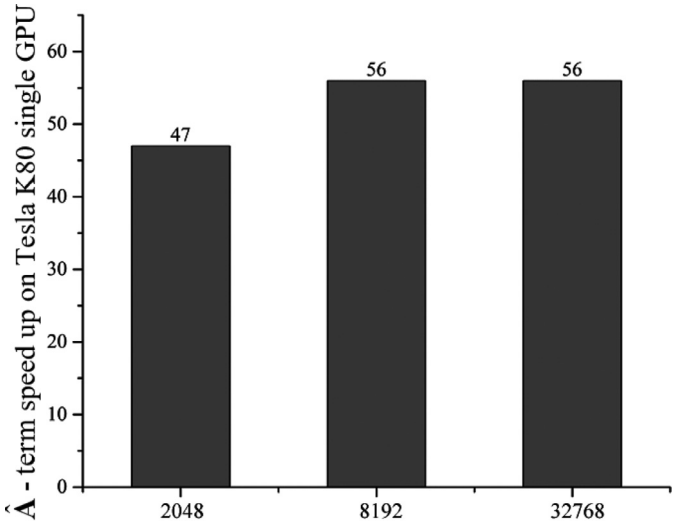

Fig. 14. Speedup achieved in evaluating the $\hat{A} - term$ in Eq. (13) on a single Tesla K80 card containing 2496 cores as a function of problem sizes.

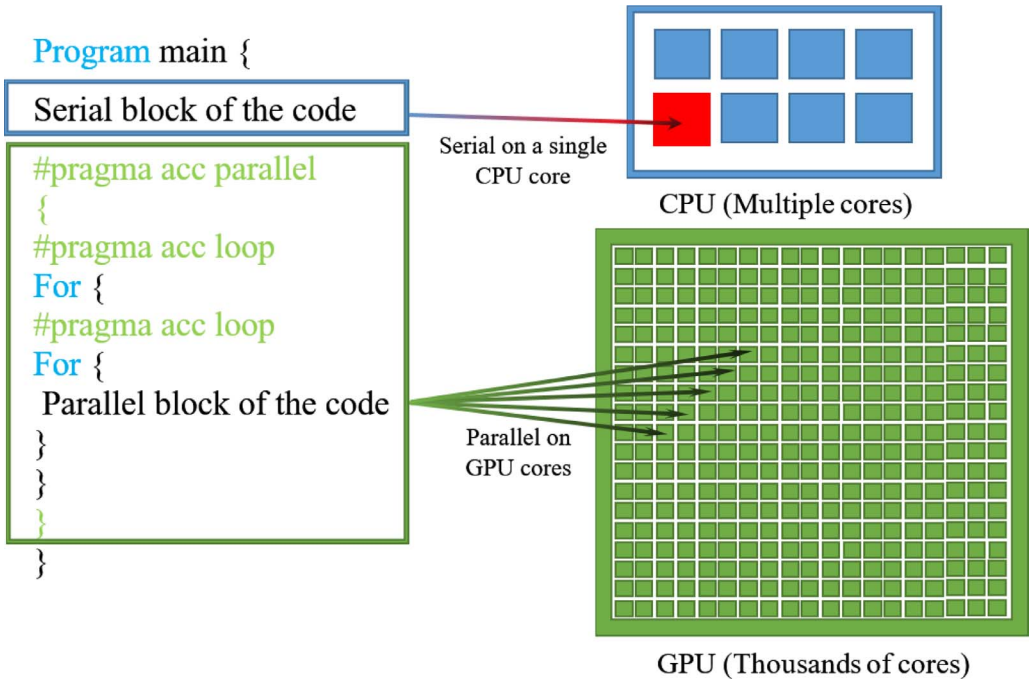

Fig. 13. Schematic of parallel GPU execution using OPENACC loop directive pragmas. The left blue box shows the serial region which runs on a CPU with multiple cores and the left green box represents the GPU section running on thousand GPU cores. The "acc parallel loop" directive pragmas around the nested loop port them to run on GPU. (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

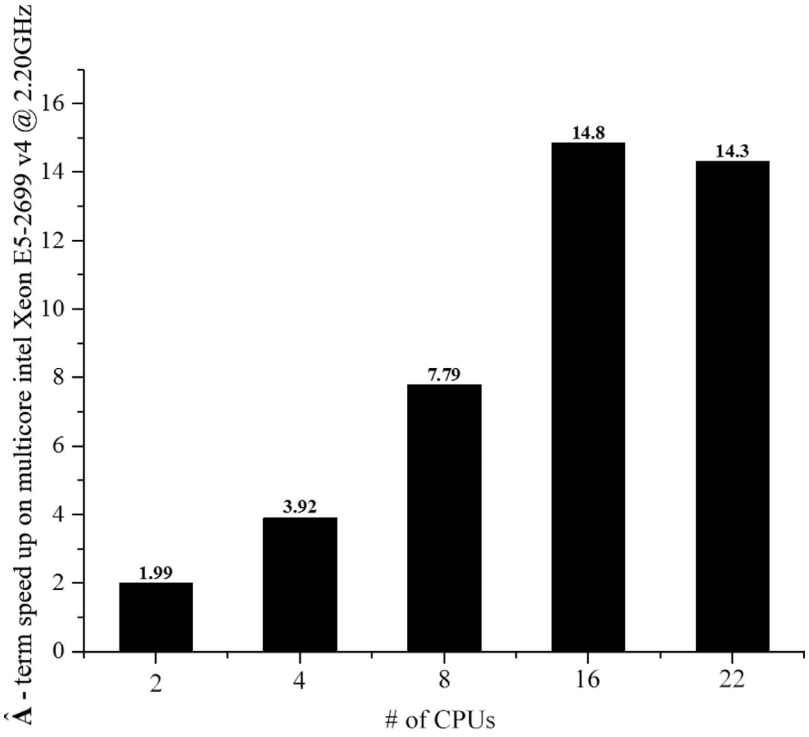

Fig. 15. Speedup of the calculation involved in evaluating the $\hat{A} - term$ in Eq. (13) as a function of CPUs in Intel Xeon CPU E5-2699 v4 clocked at 2.20 GHz for the problem size of $32 \times 2 \times 32$ voxels This is notable that this speedup will be decreased for larger problem sizes due to memory bound and shared memory access limitation. The slight decrease from 16 to 22 CPUs is due to the same reason.

computation times on the GPU, Fig. 16 shows that the GPU performance decreases with increasing problem size. Note that the FOURN subroutine was used for the FFT calculations. In order to address this problem, we profiled the ACCPFDD algorithm, which already has the hotspot subroutine running on the GPU. This is viable with PGPROF 16.0 and later because PGI has merged recently the NVIDIA visual profiler tool (i.e. NVIDIA CUDA profiler package) with PGPROF to enable profiling both the GPU and CPU parts of the code at the same time.

We can predict the expected theoretical speedup of a code using the Amdahl's Law [52]. If a fraction of code f is accelerated p times, then the net speed up of the whole code can be expressed as following:

$$S = \frac{1}{\left(1 - f + \frac{f}{p}\right)}. \tag{21}$$

According to Amdahl's law, even if we infinitely speed up the $\hat{A} - term$ for the problem size of $32b \times 2b \times 32b$, considering the fact that $\hat{A} - term$ takes 94% of the code (i.e. according to Fig. 11), the ideal speed up is 16.66x. After careful examination of the profiler results, we noticed that while we were able to minimize the time required to calculate the interaction matrix and the first term in Eq. (13) using the GPU, the FFT routines called in six different places throughout the code at each time step were now the new hotspot in the code. More importantly, FFT computation time was growing with increasing problem size. This explains the observation made previously in Fig. 11, where the percentage of time designated for calculation of the first term in Eq. (13) decreased from 94% to 90% with increasing domain size. Fig. 17 depicts the FFT runtime distribution variation with problem size in the ACCPFDD code.

Consequently, we decided to minimize the time spent on FFT routines by first considering available alternatives. Although the ultimate goal is to run the FFT on the GPU, in the next section we perform a study in which the default FFT routine used originally in the code (i.e. the standard

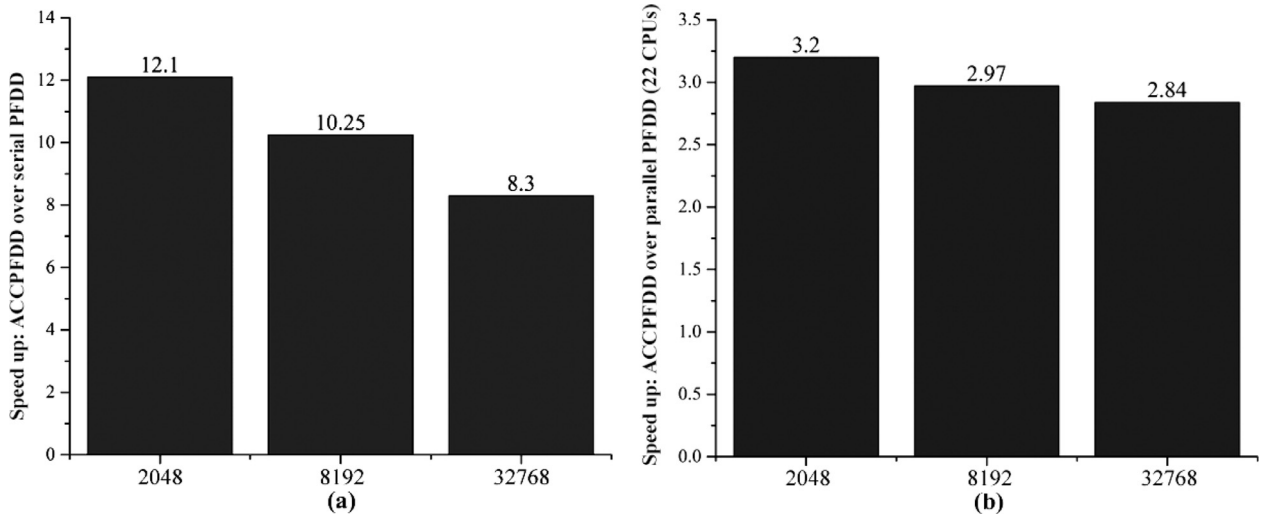

Fig. 16. Speedup: (a) ACCPFDD compared to the serial PFDD code and (b) ACCPFDD compared to the parallel PFDD code run on 22 Intel Xeon E5 cores (CPUs). Note that the FOURN subroutine was used for the FFT calculations.

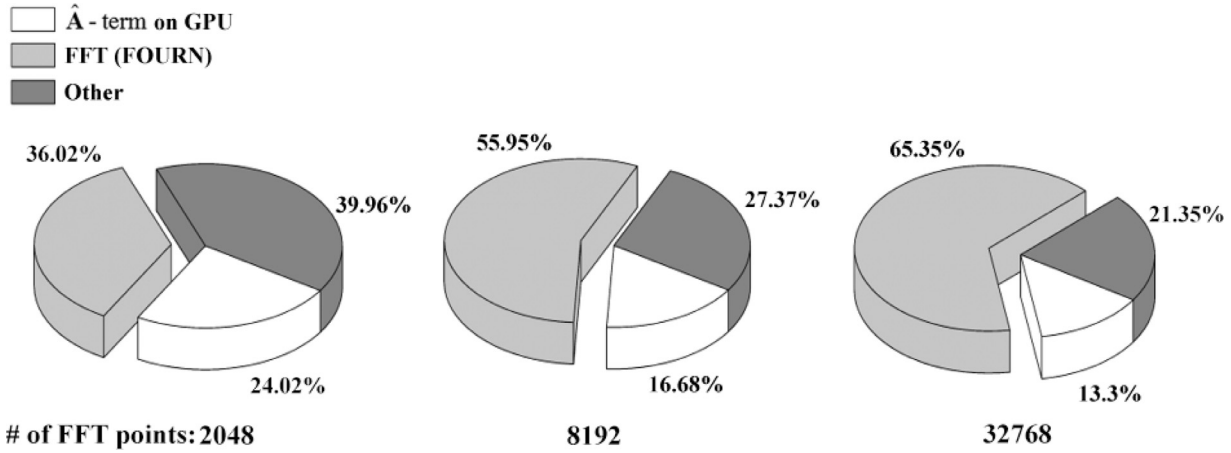

Fig. 17. Distribution of calculations for different problem sizes after porting the hotspot subroutine onto a GPU, which is the ACCPFDD code. As the legend shows, the FOURN subroutine was used for the FFT calculations.

FOURN routine from ″Numerical Recipes in C″ [45]), the FFTW library [53], and the CUFFT library (i.e. GPU version of FFT library running on CUDA) have been compared for their efficiency within ACCPFDD.

5. Performance comparison for FFT calculations: FOURN vs. FFTW vs. CUFFT

One common FFT solver used in many scientific codes is the FOURN routine that has been taken from *Numerical Recipes in FORTRAN/C* [45,54]. Although this routine is fast, recently more advanced libraries have been developed to perform FFTs in the most efficient way. Among them, FFTW [55–59] and CUFFT [60–62] are efficient FFT libraries running on CPU/Multicore CPUs and GPUs, respectively. FFTW is an efficient C FFT library developed at MIT for computation of discrete Fourier transforms (DFT) in 1D, 2D, and 3D space and uses O (N log N) algorithms to perform FFT calculations. It supports both real and complex data as raw input with arbitrary size transformations. In the presented research, we used FFTW 3.3.6 (FFTW3), which is the latest release of FFTW. FFTW3 supports all the SSE/SSE2/AVX/ARM instructions [63].

CUFFT is the NVIDIA CUDA FFT library, which was developed after FFTW. This library is able to speed up the FFT computations up to 10X on the thousand cores of a GPU in comparison to FFT computations run on CPUs. CUFFT is also able to perform transformations in all dimensions (i.e. 1D, 2D, and 3D) with both real and complex data types and large data sizes of up to 128 million elements. CUFFT is included as a part of the NVIDIA CUDA toolkit [64]. We used CUDA toolkit 7.5 for our simulations. Appendix C represents the FFTW and CUFFT structure and the way these libraries are called in the scientific codes to perform fast Fourier transforms.

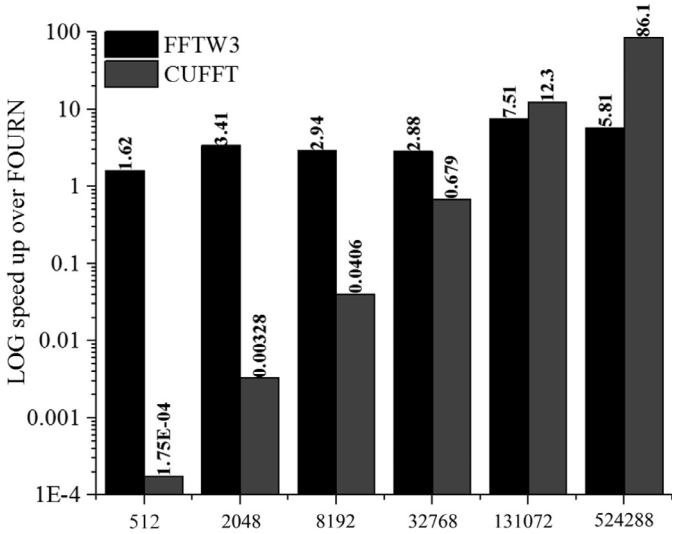

Fig. 18. Speedup of standalone FFTW3 and CUFFT over the FOURN routine for different number of FFT points (i.e. various problem sizes).

In order to benchmark the performance of these FFT solvers, we performed a study comparing standalone FFTs with a complex data type input. Fig. 18 illustrates this benchmark comparison for different problem sizes for 3D transformations. From this figure, we can understand that for all problem sizes, the FFTW library outperforms the default FOURN routine taken from Numerical Recipes. Moreover, starting from the problem size of 256 cubed (i.e. 256 × 256 × 256 FFT points), the CUFFT library outperforms the FFTW library, which is expected due to the fact that the GPU performs much better than the CPUs on large data sets and with massively parallel tasks. It is also worth mentioning that the most of time of the CUFFT calls are spent on data transfer from main memory to GPU memory. If we eliminate the amount data transfer by keeping data resident on the GPU, a significant performance gain would be obtained. This is more feasible when using OPENACC instead of CUDA to perform the data transfer (i.e. CUDA Memcpy) and the reason is that data management using OpenACC directive pragmas gives us more options while more intuitive to implement (e.g. unstructured “enter/extit data” introduced in OpenaACC 2.0 [51]). We will elaborate on such data interoperability in detail in the next section.

In order to maintain performance portability, we decided to make the code interoperable with all FFT options. This will enable us to run the code using FFTW on CPUs in the case there is no access to a GPU on a given machine/workstation, and if FFTW is not available, it can fall back to the included FOURN from Numerical Recipes. To this end, the C preprocessor ′ #ifdef ′ statements are used to switch the FFT library from CUFFT to FFTW or FOURN at compile time as needed. Fig. 19 illustrates the comparison of ACCPFDD (i.e. PFDD with the hotspot subroutine running on the GPU using OPENACC) using the three different types of FFT algorithms (i.e. FOURN, FFTW, and CUFFT). We use the terminology ACCPFDD-CUFFT to designate the PFDD code that utilizes the CUDA FFT library running on a GPU coupled with the hotspot subroutine also running on a GPU using OPENACC. Similarly, ACCPFDD-FOURN and ACCPFDD-FFTW utilize the GPU only for the hotspot subroutine, and the FFT is completed with FOURN (serial computation) or FFTW (parallel FFT computation on CPUs), respectively. Having these options allows us to run the smaller problem sizes using FFTW (i.e. because according to the Fig. 18, for smaller problem sizes, FFTW is more efficient than CUFFT) and in the case of having no access to FFTW libraries on a workstation, using FOURN. For the large

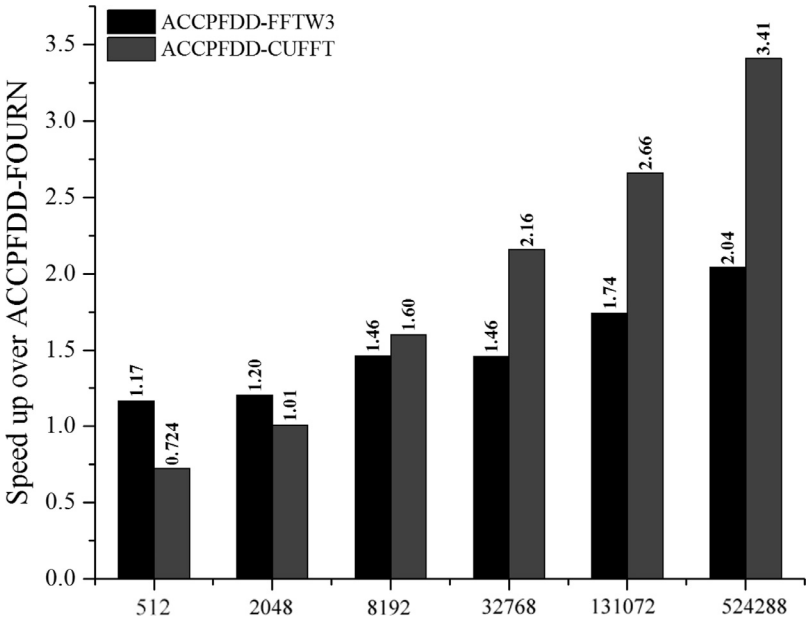

Fig. 19. Performance gain for ACCPFDD-FFTW3 and ACCPFDD-CUFFT versus ACCPFDD-FOURN for different problem sizes.

enough domains, the code switches to CUFFT to take the advantage of much more efficient FFT calculations for massive data sets. Fig. 19 tells us that the ACCPFDD-CUFFT is the fastest code in comparison to ACCPFDD-FOURN or ACCPFDD-FFTW, for the problem sizes of 8192 ($64 \times 2 \times 64$) and larger. With increasing domain size, the difference between ACCPFDD-CUFFT and ACCPFDD-FFTW becomes more significant due to the massively parallel potential of the large dataset and computations favored by the GPU architecture. It is also notable that we should not directly compare Fig. 19 with Fig. 18. In Fig. 19 we are reporting the total efficiency of PFDD code, which includes 6 calls to the FFT routine (4 forward and 2 backward transformations) each iteration in the code. Conversely, Fig. 18 shows the comparison for the standalone FFT routines with a given input to perform a single forward transformation.

6. OPENACC-CUDA interoperability

When CUDA libraries (e.g. CUFFT, CUBLAS, etc.) are called in a code, the CUDA data transfer (i.e. CUDA Memcpy) is inherited by default from the CUDA language. This fact conflicts with our ACCPFDD code running on OPENACC in two ways. Our goal is to maintain performance portability as much as possible; hence, having one part of the algorithm (i.e. the hotspot subroutine) using OPENACC and the other parts (i.e. FFT routines) using CUDA functions requires special attention to maintain consistency. Additionally, CUDA built-in functions necessitate including the "cuda.h" library header that works only with the PGI C++ (i.e. pgc++) compiler. Currently, the PFDD code is written in C, hence it would need to be fully converted to C++ would not be an inefficient use of time. Rather, with the advent of OPENACC-CUDA interoperability [65], calling CUDA functions in OPENACC data regions

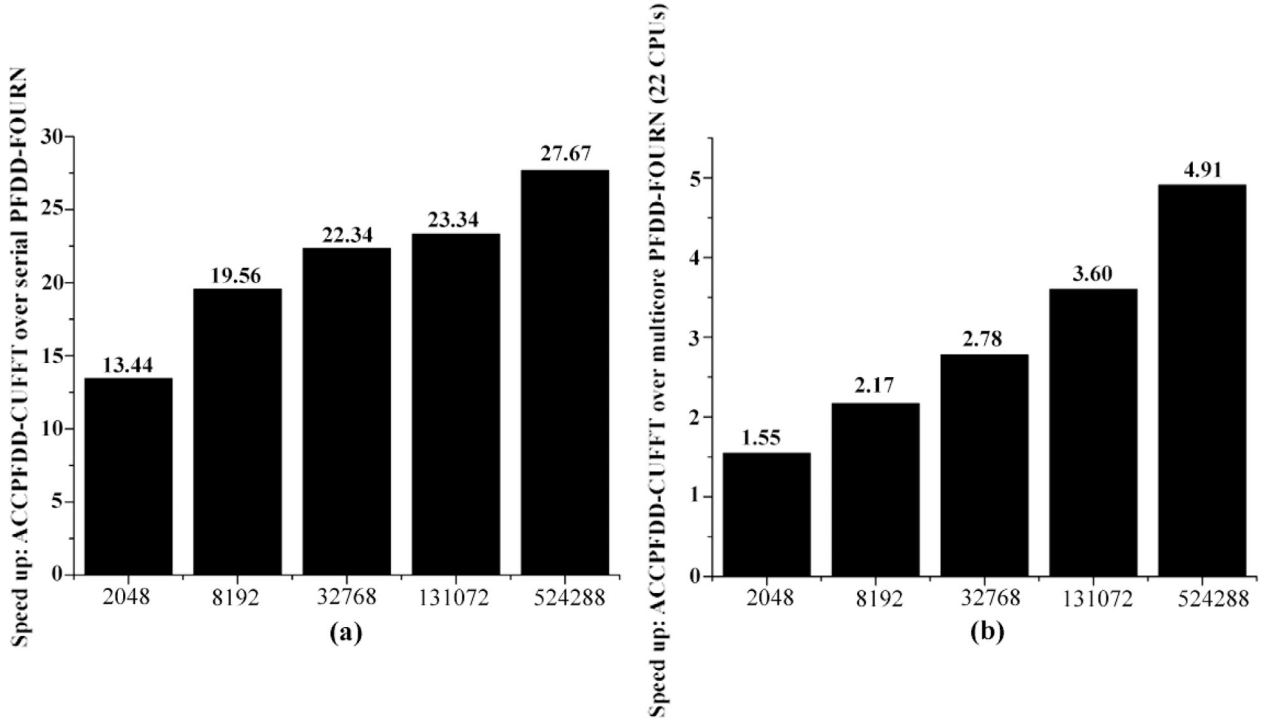

Fig. 20. Speedup: (a) ACCPFDD-CUFFT versus serial PFDD-FOURN and (b) ACCPFDD-CUFFT versus parallel PFDD-FOURN (22 CPUs) as a function of problem sizes.

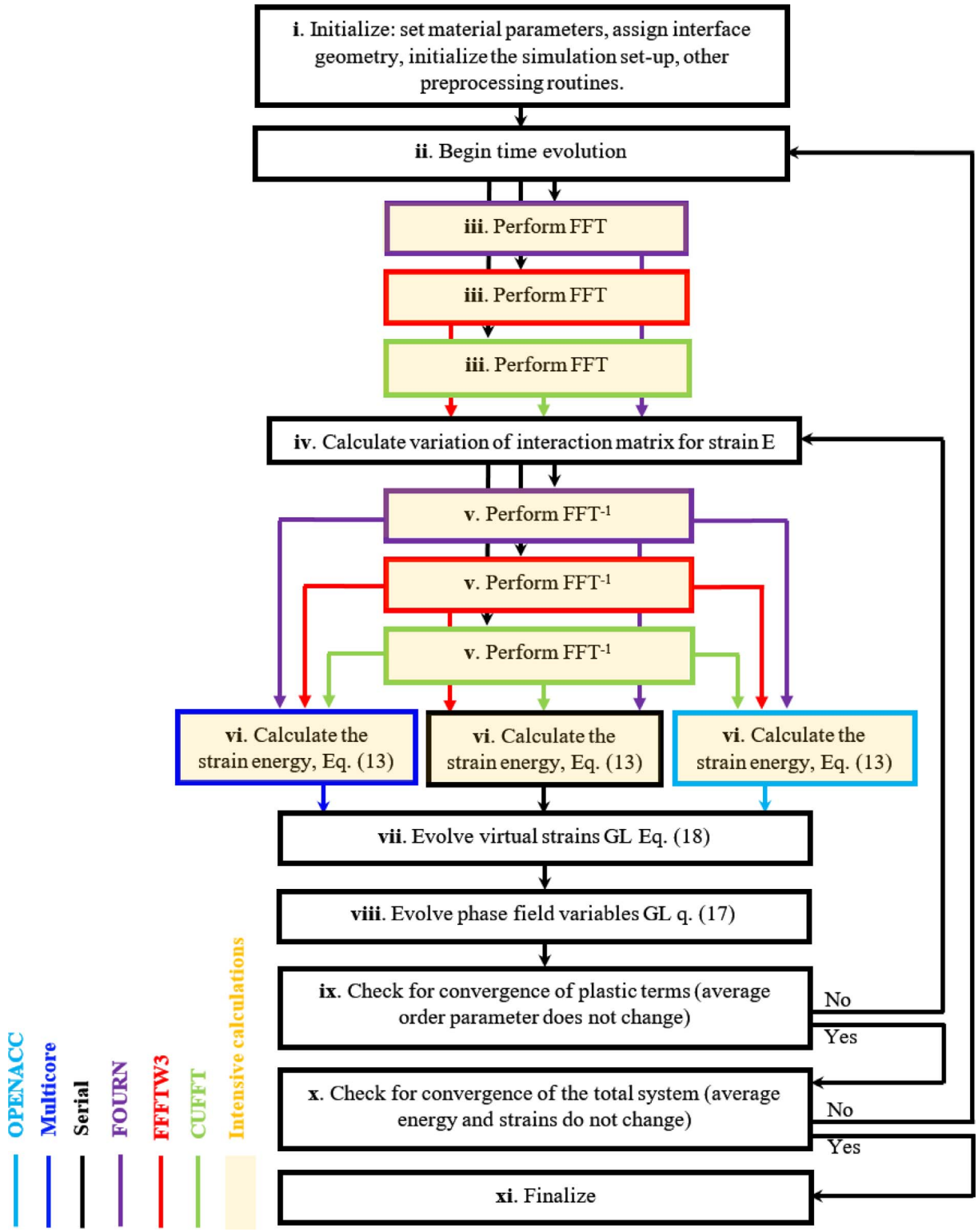

Fig. 21. Schematic of the different PFDD solvers: serial PFDD, multicore PFDD (22 CPU Intel Xeon E5), and ACCPFDD used with three different FFT calculation options FOURN, FFTW3, and CUFFT. The OPENACC region (ACCPFDD) and the CUFFT, both run on NVidia Tesla K80 GPU.

is viable. Appendix D presents the OPENACC-CUDA interoperability used for calling the CUFFT library in ACCPFDD code. The OPENACC construct ″host_data use_device (array)″, makes the address of device (i.e. GPU) data present on the host (i.e. CPU) and enables it to pass the data to the functions that use CUDA device pointers. After reading the ″use_device″ pragma, the compiler can use the device version of the array instead of the host copy. Indeed, while this is a CPU entry point (i.e. which is called in the CPU region of the code without any OPENACC pragmas or CUDA kernels), it invokes GPU kernels that act directly on the GPU memory.

Using the OPENACC-CUDA interoperability, we are able to use OPENACC data directive pragmas for CPU-GPU data management with much more flexibility because the OPENACC data regions are easier and far more flexible to control comparing to the CUDA Memcopy functions. This becomes obvious when using unstructured OPENACC data regions [51] in which, ″ACC enter/exit data" as a heterogeneous data

structures can be implemented anywhere in the code.

By incorporating the OPENACC-CUDA interoperability for calling the CUFFT library in the ACCPFDD code, a final version of ACCPFDD-CUFFT was developed. Fig. 20 illustrates the final speed up comparing the ACCPFDD-CUFFT running on a single Tesla K80 GPU to both serial and multicore versions of the code using the original FOURN routine running on Intel Xeon CPU E5-2995 v4 @ 2.20 GHz with 22 cores.

With a careful evaluation of Fig. 20, we see that the issue of decreasing GPU efficiency with increasing problem size addressed earlier in Fig. 16 has now has been resolved. After minimizing the FFT routines on the GPU using the CUFFT library together with the OPENACC_CUDA interoperability, we gained up to 27.67x speed up comparing to the serial version and 5x speed up relative to the multicore code running on 22 cores. The increasing trend in scalability asserts the efficient implementation of GPU for acceleration of the PFDD code. It is also notable that not only does the scalability grow with problem size, the ratio of speedup for two subsequent problem sizes is also being increased. This justifies the computational efficiency of the developed ACCPFDD-CUFFT code as a promising accelerated tool to run massively parallel dislocation dynamics simulations on graphics hardware units.

To provide an overall understanding, the flow chart in Fig. 21 describes the flowchart for serial PFDD, parallel CPU version, and the ACCPFDD code all having the options to run with different FFT solvers FOURN, FFTW3, and CUFFT. The highlighted region in the schematic shows the hotspot regions of the code ported to GPU.

7. Conclusion

The first implementation of the phase field dislocation dynamics (PFDD) algorithm on a GPU has been presented here. The OPENACC standard together with CUFFT library and OPENACC-CUDA interoperability have been exploited to accelerate the PFDD OPENACC-CUFFT (ACCPFDD-CUFFT) code on a single Tesla K80 up to 27.6x in comparison to the serial version and up to 5x in comparison to the 22 core Intel Xeon CPU E5-2699 v4 @ 2.20 GHz. Moreover, a comprehensive study has been conducted comparing the performance benchmark of available standard FFT routines and libraries including FOURN, FFTW, and CUFFT as a CUDA FFT library. Growing rate of ACCPFDD-CUFFT simulation speedup on a larger problem size justifies the efficient implementation of the code on GPU. The ACCPFDD-CUFFT code has been validated using the analytical solution available for the stress field around an infinite edge dislocation. Moreover, a Cu–Ni bi-crystal interface system has been simulated and the results are compared using serial/multicore/GPU version of the code to assure identical results are produced. The GPU accelerated PFDD code, provides insight to solve a wide variety of computationally intensive problems in which dislocation motion and configurations are determining factors in the material response. In future work, we plan to port the ACCPFDD-CUFFT on multiple GPUs using the MPI (Massage passing interface) standard in order to achieve even higher speedup for the phase field dislocation dynamics solver.

Acknowledgments

This work is based upon a project supported by the U.S. National Science Foundation (NSF) under grant no. CMMI-1650641. This support is gratefully acknowledged. A. H. would like to acknowledge support from the Los Alamos National Laboratory Directed Research and Development (LDRD) Program under the Project # 20160156ER. I.J.B. acknowledges financial support from NSF under grant no. CMMI-1728224.

Appendix A. (Loop intensities in the energy calculation routine as the PFDD hotspot)

```
for(k1=0;k1<N1;k1++)
    {/*Intensity : 385.34*/
      for(k2=0;k2<N2;k2++)
         {/*Intensity : 380.49*/
          for(k3=0;k3<N3;k3++)
             {/*Intensity : 375.61
               Calculate frequencies
               for (m=0; m<ND; m++)
                     {/*Intensity : 209.22*/
                    for (n=0; n<ND; n++)
                        {/*Intensity : 152.51*/
                        for (u=0; u<ND; u++)
                          {/*Intensity : 84.11*/
                            for (v=0; v<ND; v++)
                                {/*Intensity : 57.11*/

                           for (i=0; i<ND; i++) {/*Intensity : 32.06*/
                             for (j=0; j<ND; j++) {/*Intensity : 19.00*/
                               for (k=0; k<ND; k++) {/*Intensity : 8.14*/
                                for (l=0; l<ND; l++) {/*Intensity : 4.00*/

                                  Calculate the energy terms
                                              }
                                         }
                                    }
                                }

                             }
                         }
                    }

             }
         }
     }
```

Appendix B. (Energy_Calc hotspots on GPU using OPENACC)

Part A:

```
1     #pragma acc enter data
copyin(fx[0:N1*N2*N3],fy[0:N1*N2*N3],fz[0:N1*N2*N3])  create(A)
create(E_real)
2     #pragma acc kernels
3     {
4     #pragma acc loop collapse (3) gang (512)
private(k1,k2,k3,m,n,u,v,i,j,k,l,fk,nfreq,fk4,G)
5     for(k1=0;k1<N1;k1++)
6     {
7         for(k3=0;k3<N3;k3++)
8         {
9            for(k2=0;k2<N2;k2++)
10            {
11
12                    Calculate frequencies
13    #pragma acc loop vector private(G) collapse (4)
14                        for (m=0; m<ND; m++) {
15                            for (n=0; n<ND; n++) {
16                                for (u=0; u<ND; u++) {
17                                    for (v=0; v<ND; v++) {
18
19    #pragma acc loop private(G) collapse (3)
20                                        for (i=0; i<ND; i++) {
21                                            for (j=0; j<ND; j++) {
22                                                for (k=0; k<ND; k++) {
23
24                                                    for (l=0; l<ND; l++) {

                                                    Calculate the energy terms
25                                                   }
26                                                }
27                                            }
```

Part B:

```
1   #pragma acc kernels
2   {
3    #pragma acc loop gang  collapse(3)
4   for (k1=0; k1<N1; k1++) {
5     for (k3=0;k3<N3;k3++) {
6        for (k2=0;k2<N2;k2++) {
7
8    #pragma acc loop vector  collapse(4)
9          for (u=0; u<ND; u++) {
10               for (v=0; v<ND; v++) {
11                for (m=0; m<ND; m++) {
12                    for (n=0; n<ND; n++) {
13
14                    Calculate the energy terms
15                            }
16                        }
17                    }
18                }
19
20            }
21        }
22    }
23  } //#pragma acc kernels
24  #pragma acc kernels
25  {
26   en = 0.;
27  #pragma acc loop collapse(3)
28   for (i=0; i<N1; i++) {
29            for (k=0; k<N3; k++) {
30          for (j=0; j<N2; j++) {
31
32
                    Calculate the energy terms
33            }
34        }
35    }

36  }//#pragma acc kernels
37#pragma acc exit data delete(fx[0:N1*N2*N3],fy[0:N1*N2*N3],fz[0:N1*N2*N3])
delete(A)  delete (E_real)
```

Appendix C. (FFTW and CUFFT library calls)

FFTW

```
static fftw_complex signal[NUM_POINTS];
static fftw_complex result[NUM_POINTS];
fftw_plan plan = fftw_plan_dft_1d(NUM_POINTS,  /*Create plan*/
                 signal,
                 result,
                 FFTW FORWARD,
                 FFTW_ESTIMATE);
acquire_from_somewhere(signal);               /*Input data*/
fftw_execute(plan);                           /*Perform FFT*/
do something with(result);                    /*Ouput data*/
fftw destroy plan(plan);                      /*Free memory*/
```

CUFFT

```
cufftHandle plan;
cufftComplex *data1, *data2;
cudaMalloc((void**)&data1, sizeof(cufftComplex)*NX*NY*NZ);
cudaMalloc((void**)&data2, sizeof(cufftComplex)*NX*NY*NZ);
/* Create a 3D FFT plan*/
cufftPlan3d(&plan, NX, NY, NZ, CUFFT_C2C);
/* Transform the first signal in place*/
cufftExecC2C(plan, data1, data1, CUFFT FORWARD);
/* Transform the second signal using the same plan*/
cufftExecC2C(plan, data2, data2, CUFFT_FORWARD);
/* Destroy the cuFFT plan. */
cufftDestroy(plan);
cudaFree(data1); cudaFree(data2);
```

Appendix D. (OPENACC-CUDA interoperability in calling the CUFFT CUDA library)

CUDA in CUFFT

```
typedef float fftw_complex[2];
fftw_complex *vx = new fftw_complex[NX*NY*NZ];
float *d vx;
cudaMalloc(&d_vx, NX*NY*NZ*sizeof(fftw_complex));
cudaMemcpy(d vx, vx, NX*NY*NZ*sizeof(fftw complex),
cudaMemcpyHostToDevice);
cufftHandle planc2c;
cufftPlan3d(&planc2c, NZ,NY, NX, CUFFT_C2C);
cufftSetCompatibilityMode(planc2c, CUFFT_COMPATIBILITY_NATIVE);
cufftExecC2C(planc2c, (cufftComplex *)d_vx, (cufftComplex *)d_vx,
CUFFT FORWARD);
cudaMemcpy(vx, d_vx, NX*NY*NZ*sizeof(fftw_complex),
cudaMemcpyDeviceToHost);
```

OPENACC-CUDA interoperability in CUFFT

```
typedef float fftw_complex[2];
fftw_complex *vx = new fftw_complex[NX*NY*NZ];
 #pragma acc data copy(vx[0:NX*NY*NZ][0:1])
 }
cufftHandle planc2c;
cufftPlan3d(&planc2c, NZ,NY, NX, CUFFT_C2C);
cufftSetCompatibilityMode(planc2c, CUFFT_COMPATIBILITY_NATIVE);
 #pragma acc host_data use_device(vx)
cufftExecC2C(planc2c, (cufftComplex *)d_vx, (cufftComplex *)d_vx,
CUFFT_FORWARD);
{ /*end pragam acc data*/
```

References

[1] Zhu Y. Special issue: nanostructured materials preface. Spring St, New York: Springer; 2013. p. 233. NY 10013 USA.
[2] Valiev R. Nanostructuring of metals by severe plastic deformation for advanced properties. Nat Mater. 2004;3. 1476–122.
[3] Zhu YT, Liao X. Nanostructured metals: retaining ductility. Nat Mater 2004;3:351–2.
[4] Carpenter JS, Nizolek T, McCabe RJ, Knezevic M, Zheng SJ, Eftink BP, et al. Bulk texture evolution of nanolamellar Zr–Nb composites processed via accumulative roll bonding. Acta Mater 2015;92:97–108.
[5] Ardeljan M, Savage DJ, Kumar A, Beyerlein IJ, Knezevic M. The plasticity of highly oriented nano-layered Zr/Nb composites. Acta Mater 2016;115:189–203.
[6] Ardeljan M, McCabe RJ, Beyerlein IJ, Knezevic M. Explicit incorporation of deformation twins into crystal plasticity finite element models. Comput Methods Appl Mech Eng 2015;295:396–413.
[7] Zecevic M, Knezevic M. A dislocation density based elasto-plastic self-consistent model for the prediction of cyclic deformation: Application to Al6022-T4. Int J Plast 2015;72:200–17.
[8] Zecevic M, Korkolis YP, Kuwabara T, Knezevic M. Dual-phase steel sheets under cyclic tension–compression to large strains: Experiments and crystal plasticity modeling. J Mech Phys Solids 2016;96:65–87.
[9] Jahedi M, Paydar MH, Zheng S, Beyerlein IJ, Knezevic M. Texture evolution and enhanced grain refinement under high-pressure-double-torsion. Mater Sci Eng A 2014;611:29–36.
[10] Xu S, Guo Y, Ngan A. A molecular dynamics study on the orientation, size, and dislocation confinement effects on the plastic deformation of Al nanopillars. Int J Plast 2013;43:116–27.
[11] Anderson JA, Lorenz CD, Travesset A. General purpose molecular dynamics simulations fully implemented on graphics processing units. J Comput Phys 2008;227:5342–59.
[12] Baker JA, Hirst JD. Molecular dynamics simulations using graphics processing units. Molecular Inf 2011;30:498–504.
[13] Graham J, Rollett A, LeSar R. Fast Fourier transform discrete dislocation dynamics. Model Simul Mater Sci Eng 2016;24:085005.
[14] Bulatov VV, Tang M, Zbib HM. Crystal plasticity from dislocation dynamics. MRS Bull 2011;26:191–5.
[15] Bulatov VV, Hsiung LL, Tang M, Arsenlis A, Bartelt MC, Cai W, et al. Dislocation multi-junctions and strain hardening. Nature 2006;440:1174–8.
[16] Tarleton E. Dislocations, Mesoscale Simulations and Plastic Flow. Oxford series on materials modelling 5, by ladislas Kubin: Scope: textbook. Level: postgraduate, researcher. Taylor & Francis; 2013.
[17] LeSar R. Simulations of dislocation structure and response. Annu Rev Condens Matter Phys. 2014;5:375–407.
[18] Bertin N, Upadhyay M, Pradalier C, Capolungo L. A FFT-based formulation for efficient mechanical fields computation in isotropic and anisotropic periodic discrete dislocation dynamics. Model Simul Mater Sci Eng 2015;23:065009.
[19] Beyerlein I, Hunter A. Understanding dislocation mechanics at the mesoscale using phase field dislocation dynamics. Phil Trans R Soc A 2016;374:20150166.
[20] Zeng Y, Hunter A, Beyerlein IJ, Koslowski M. A phase field dislocation dynamics model for a bicrystal interface system: An investigation into dislocation slip transmission across cube-on-cube interfaces. Int J Plast 2016;79:293–313.
[21] Bauer P, Klement V, Oberhuber T, Žabka V. Implementation of the Vanka-type multigrid solver for the finite element approximation of the Navier–Stokes equations on GPU. Comput Phys Commun 2016;200:50–6.
[22] Dolbeau, R. Theoretical peak FLOPS per instruction set on modern Intel CPUs. 2015.
[23] Cecka C, Lew AJ, Darve E. Assembly of finite element methods on graphics processors. Int J Numer Methods Eng 2011;85:640–69.
[24] Dziekonski A, Sypek P, Lamecki A, Mrozowski M. Generation of large finite-element matrices on multiple graphics processors. Int J Numer Methods Eng 2013;94:204–20.
[25] Savage DJ, Knezevic M. Computer implementations of iterative and non-iterative crystal plasticity solvers on high performance graphics hardware. Comput Mech 2015;56:677–90.
[26] Mellbin Y, Hallberg H, Ristinmaa M. Accelerating crystal plasticity simulations using GPU multiprocessors. Int J Numer Methods Eng 2014;100:111–35.
[27] Mihaila B, Knezevic M, Cardenas A. Three orders of magnitude improved efficiency with high-performance spectral crystal plasticity on GPU platforms. International Journal for Numerical Methods in Engineering 2014;97:785–98.
[28] Knezevic M, Savage DJ. A high-performance computational framework for fast crystal plasticity simulations. Comput Mater Sci 2014;83:101–6.
[29] Knezevic M, Savage DJ, Landry NW. Towards computationally tractable simulations of metal forming processes with evolving microstructures. ASME 2014 international manufacturing science and engineering conference collocated with the JSME 2014 international conference on materials and processing and the 42nd North American manufacturing research conference. American Society of Mechanical Engineers; 2014. V002T02A70–VT02A70.
[30] Chen L-Q. Phase-field models for microstructure evolution. Annu Rev Mater Res 2002;32:113–40.
[31] Lei L, Marin JL, Koslowski M. Phase-field modeling of defect nucleation and propagation in domains with material inhomogeneities. Model Simul Mater Sci Eng 2013;21:025009.
[32] Koslowski M, Cuitino AM, Ortiz M. A phase-field theory of dislocation dynamics, strain hardening and hysteresis in ductile single crystals. J Mech Phys Solids 2002;50:2597–635.
[33] Mura T. Micromechanics of defects in solids. Springer Science & Business Media; 2013.
[34] Mura, T. The continuum theory of dislocations. 1968.
[35] Hoagland R, Mitchell T, Hirth J, Kung H. On the strengthening effects of interfaces in multilayer fee metallic composites. Philosoph Mag A 2002;82:643–64.
[36] Hoagland RG, Kurtz RJ, Henager CH. Slip resistance of interfaces and the strength of metallic multilayer composites. Scr Mater 2004;50:775–9.
[37] Wang YU, Jin YM, Khachaturyan AG. Phase field microelasticity modeling of dislocation dynamics near free surface and in heteroepitaxial thin films. Acta Mater 2003;51:4209–23.
[38] Wang YU, Jin Y, Cuitino A, Khachaturyan A. Nanoscale phase field microelasticity theory of dislocations: model and 3D simulations. Acta Mater 2001;49:1847–57.
[39] Koslowski M, Lee DW, Lei L. Role of grain boundary energetics on the maximum strength of nanocrystalline nickel. J Mech Phys Solids 2011;59:1427–36.
[40] Ortiz M, Phillips R. Nanomechanics of defects in solids. Adv Appl Mech 1998;36:1–79.
[41] Hunter A, Beyerlein I. Relationship between monolayer stacking faults and twins in nanocrystals. Acta Mater 2015;88:207–17.
[42] Hunter A, Beyerlein I. Stacking fault emission from grain boundaries: material dependencies and grain size effects. Mater Sci Eng 2014;600:200–10.
[43] Hirth, J.P. Theory of dislocations. 1968.
[44] Srolovitz D, Lomdahl P. Dislocation dynamics in the 2-d Frenkel–Kontorova model. Physica D 1986;23:402–12.
[45] Press WH, Teukolsky SA, Vetterling WT, Flannery BP. Numerical recipes in c. Cambridge University Press; 1982.
[46] Portland Group, PGI Compiler User's Guide for Intel 64 and AMD64C PUs. 2016.
[47] Zherebtsov S, Murzinova M, Salishchev G, Semiatin S. Spheroidization of the lamellar microstructure in Ti–6Al–4V alloy during warm deformation and annealing. Acta Mater 2011;59:4138–50.
[48] Chen W, Wang X, Hu L, Wang E. Fabrication of ZK60 magnesium alloy thin sheets with improved ductility by cold rolling and annealing treatment. Mater Des 2012;40:319–23.
[49] Lebacki B, Wolfe M, Miles D. The pgi fortran and c99 openacc compilers. Cray User Group; 2012.
[50] Herdman J, Gaudin W, McIntosh-Smith S, Boulton M, Beckingsale DA, Mallinson A, et al. Accelerating hydrocodes with OpenACC, OpenCL and CUDA. High performance computing, networking, storage and analysis (SCC), 2012 SC companion. IEEE; 2012. p. 465–71.
[51] Farber, R. Parallel programming with OpenACC: Newnes; 2016.
[52] Hill MD, Marty MR. Amdahl's law in the multicore era. Computer 2008;41.
[53] Koplik J, Needleman A. Void growth and coalescence in porous plastic solids. Int J Solids Struct 1988;24:835–53.
[54] William H. Numerical recipes in fortran. Cambridge University Press; 1992.
[55] Frigo M, Johnson SG. FFTW: An adaptive software architecture for the FFT. Acoustics, speech and signal processing, 1998 proceedings of the 1998 IEEE international conference on: IEEE. 1998. p. 1381–4.
[56] Frigo M, Johnson SG. FFTW user's manual. Massachusetts Institute of Technology; 1999.
[57] Frigo, M., Johnson, S.G. Fastest {F} ourier {T} ransform in the {W} est. 2006.
[58] Frigo M, Johnson SG. The fastest fourier transform in the west. DTIC Doc 1997.
[59] Frigo M, Johnson SG. FFTW: fastest Fourier transform in the west. Astrophys Source Code Lib 2012.
[60] Nvidia, C. CUFFT library. Version; 2010.
[61] Nugmanov DR, Sitdikov OS, Markushev MV. Microstructure evolution in MA14 magnesium alloy under multi-step isothermal forging. Lett Mater 2011;1:213–6.
[62] Nvidia, C. Compute unified device architecture programming guide. 2007.
[63] Dolbeau, R. Theoretical peak FLOPS per instruction set on less conventional hardware. 2015.
[64] Nvidia C. Toolkit documentation. NVIDIA CUDA getting started guide for linux. 2014.
[65] Withers P, Bhadeshia H. Residual stress. Part 1–measurement techniques. Mater Sci Technol 2001;17:355–65.