Computer Physics Communications 254 (2020) 107231

Contents lists available at ScienceDirect

Computer Physics Communications

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

A multi-GPU implementation of a full-field crystal plasticity solver for efficient modeling of high-resolution microstructures☆

Adnan Eghtesad [a], Kai Germaschewski [b], Ricardo A. Lebensohn [c], 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] Theoretical Division, Los Alamos National Laboratory, Los Alamos, NM 87544, USA

ARTICLE INFO

Article history:
Received 17 September 2018
Received in revised form 19 February 2020
Accepted 20 February 2020
Available online 28 February 2020

Keywords:
Crystal plasticity
Microstructures
Parallel computing
GPU
OpenACC
MPI

ABSTRACT

In a recent publication (Eghtesad et al., 2018), we have reported a message passing interface (MPI)-based domain decomposition parallel implementation of an elasto-viscoplastic fast Fourier transform-based (EVPFFT) micromechanical solver to facilitate computationally efficient crystal plasticity modeling of polycrystalline materials. In this paper, we present major extensions to the previously reported implementation to take advantage of graphics processing units (GPUs), which can perform floating point arithmetic operations much faster than traditional central processing units (CPUs). In particular, the applications are developed to utilize a single GPU and multiple GPUs from one computer as well as a large number of GPUs across nodes of a supercomputer. To this end, the implementation combines the OpenACC programming model for GPU acceleration with MPI for distributed computing. Moreover, the FFT calculations are performed using the efficient Compute Unified Device Architecture (CUDA) FFT library, called CUFFT. Finally, to maintain performance portability, OpenACC-CUDA interoperability for data transfers between CPU and GPUs is used. The overall implementations are termed ACC-EVPCUFFT for single GPU and MPI-ACC-EVPCUFFT for multiple GPUs. To facilitate performance evaluation studies of the developed computational framework, deformation of a single phase copper is simulated, while to further demonstrate utility of the implementation for resolving fine microstructures, deformation of a dual-phase steel DP590 is simulated. The implementations and results are presented and discussed in this paper.

1. Introduction

Polycrystal plasticity models can be used for predicting material behavior in simulations of metal forming and evaluation of component performances under service conditions. In metal forming, material typically undergoes large plastic strains while developing spatially heterogeneous stress–strain fields [1–6]. Crystallographic slip while accommodating plastic strains induces anisotropy in the material response by evolution of texture and microstructure, which play important roles in the local and overall deformation processes of a material. Local deformation behavior can be captured using complex full-field crystal plasticity models, where the constituent grains explicitly interact with each other. The ability to perform such complex numerical simulations is recognized as a large computational challenge, because the material models must take into account a large number of physical details at multiple length and temporal scales [7–11]. Indeed, one of the main deterrent to the use of crystal plasticity theories in place of the continuum plasticity theories presently used in practice, is that the implementation of crystal plasticity theories in a full-field modeling framework requires a prohibitive increase in computational effort. This paper is concerned with the development of an efficient computational framework while emphasizing the cutting-edge, high-performance algorithms for full-field crystal plasticity models. Efficient numerical schemes at the level of the microstructural cell as a representative volume element (RVE) of a polycrystalline aggregate presented here are aimed at rendering possible the future accurate multi-level simulations of deformation in metallic materials by embedding this microstructural cell constitutive models within macro-scale finite element (FE) frameworks at each FE integration point.

Effective properties of a microstructural cell embedding crystal plasticity can be solved using finite elements with sub-grain mesh resolution [12–19]. Subsequently, these FE calculations of microstructure sensitive material behavior can be embedded within macroscopic FE model [20]. Since both the cell and macro-scale calculations are carried out simultaneously, the strategy

☆ The review of this paper was arranged by Prof. D.P. Landau.

* Correspondence to: Department of Mechanical Engineering, University of New Hampshire, 33 Academic Way, Kingsbury Hall, W119, Durham, NH 03824, USA.
E-mail address: marko.knezevic@unh.edu (M. Knezevic).

https://doi.org/10.1016/j.cpc.2020.107231
0010-4655/

is known as the FE^2 method. The FE^2 method is not practical because it is extremely computationally intensive. A Green function method has been developed as an alternative to FE for solving the field equations over a spatial microstructural cell domain [21–24]. It relies on the efficient fast Fourier transform (FFT) algorithm to solve the convolution integral representing stress equilibrium under strain compatibility constraint over a voxel-based microstructural cell, as opposed to finite element mesh. The elasto-visco plastic FFT (EVPFFT) formulation is the most advanced of several known Green's-based solvers for crystal plasticity simulations [25]. Nevertheless, numerical implementations of EVPFFT within FE would also demand substantial computational resources. Thus, acceleration of the full-field FFT-based computations is an essential task.

Several approaches involving efficient numerical schemes and high performance computational platforms have been explored to accelerate numerical procedures [26–35]. Some of the most promising approaches rely on building databases of pre-computed single crystal solutions in the form of a spectral representation [36–44], or storing the polycrystal responses calculated during the actual simulation and using them in an adaptive manner [45,46]. The single crystal solutions are used within homogenization models as well as FE full field models to represent the overall behavior of the polycrystal [41,42,47–55]. In a recent work [56], we have reported a message passing interface (MPI)-based domain decomposition parallel implementation of the EVPFFT model. The domain decomposition was performed over voxels of a microstructural cell. Moreover, we have evaluated the efficiency of several FFT libraries like FFTW [57]. Depending on the hardware at hand, significant speedups have been achieved using MPI-EVPFFTW.

In this work, we present major extensions to the previously reported implementation to take full advantage of graphics processing units (GPUs). GPUs can perform floating point arithmetic computations much faster than the traditional central processing units (CPUs). With the advent of GPUs, the era of high performance computing (HPC) has been revolutionized [58–60]. While there are many large clusters using conventional CPUs to run a job in parallel, the operating cost for running CPU-only clusters is significantly higher comparing to GPUs [61]. As an example, an ExaFLOP supercomputer operating on CPU, was estimated to demands electric power equal to the amount needed to initiate the Bay area power system [62]. GPUs are accelerators originally designed for 3D visualization and optimized for parallel processing of millions of polygons with massive data sets [63]. Hardware-wise, GPUs are much more computationally powerful than CPUs when it comes to massive parallelism. While the memory bandwidth for CPU is not more than 68 GB/s for systems with PC3-17000 DDR3 modules and quad-channel architecture, the NVIDIA Tesla K80 and Tesla P100 own memory bandwidths' of up to 480 GB/s and 720 GB/s, respectively. While the cutting-edge Intel Xeon phi processor 7250 is composed of up to 68 cores, the Tesla K80 and Tesla P100 are composed of 4992 (i.e. 2×2496) and 3584 computing cores, respectively, resulting in computing power of up to 2910 (i.e. 2×1455) and 4670 GFlops. It is notable that GPU cores (Compute Unified Device Architecture (CUDA) cores) are weaker comparing to conventional CPU cores, however, thousands of them working together will result in significantly higher computational power.

The implementation developed here combines OpenACC [64] and MPI. First, the single crystal Newton–Raphson (NR) solver is accelerated using OpenACC to run on single and multiple GPUs. The latter is MPI-OpenACC. Next, the FFT calculations are performed using the CUDA FFT library, CUFFT [65]. OpenACC-CUDA interoperability is used to control the data transfer between CPU and GPU when interfacing with native CUDA code. Finally, the remaining subroutines, except read and write, are ported to GPU for ultimate speed up gains. The overall implementation is termed ACC-EVPCUFFT for execution on a single GPU and MPI-ACC-EVPCUFFT for execution on multiple GPUs. We present speedups obtained using NVIDIA Tesla K80 and P100 GPUs relative to the original serial implementation and a recent MPI implementation of the code [56]. The proposed computational algorithms have been successfully applied to crystal plasticity modeling of pure copper and a dual-phase steel.

2. Summary of the EVPFFT model

In our notation, tensors are denoted by non-italic bold letters while scalars are italic and not bold. In the crystal plasticity framework, the viscoplastic strain rate, $\dot{\boldsymbol{\varepsilon}}^p(\mathbf{x})$ is related to the stress $\boldsymbol{\sigma}(\mathbf{x})$ at a single-crystal material point $\mathbf{x}$ through a sum over the N active slip systems, of the form [66,67]

$$\dot{\boldsymbol{\varepsilon}}^p(\mathbf{x}) = \sum_{s=1}^{N} \mathbf{m}^s(\mathbf{x})\dot{\gamma}^s(\mathbf{x}) = \dot{\gamma}_0 \sum_{s=1}^{N} \mathbf{m}^s(\mathbf{x}) \left(\frac{|\mathbf{m}^s(\mathbf{x}) : \boldsymbol{\sigma}(\mathbf{x})|}{\tau_c{}^s(\mathbf{x})} \right)^n \operatorname{sgn} \times \left(\mathbf{m}^s(\mathbf{x}) : \boldsymbol{\sigma}(\mathbf{x}) \right), \tag{1}$$

where $\dot{\gamma}^s(\mathbf{x})$, $\tau_c{}^s(\mathbf{x})$ and $\mathbf{m}^s(\mathbf{x})$ are, respectively, the shear rate, the critical resolved shear stress (CRSS) and the Schmid tensor, associated with slip system (s) at point $\mathbf{x}$. Parameter $\dot{\gamma}_0$ is a normalization factor and n is the power-law exponent representing the inverse of the material rate-sensitivity.

The Hooke's law is implemented to represent the elasto-plastic decomposition of stress–strain constitutive relation:

$$\boldsymbol{\sigma}^{t+\Delta t}(\mathbf{x}) = \mathbf{C}(\mathbf{x}) : \boldsymbol{\varepsilon}^{e,t+\Delta t}(\mathbf{x}) = \mathbf{C}(\mathbf{x}) : \left(\boldsymbol{\varepsilon}^{t+\Delta t}(\mathbf{x}) - \boldsymbol{\varepsilon}^{p,t}(\mathbf{x}) - \dot{\boldsymbol{\varepsilon}}^{p,t+\Delta t}(\mathbf{x}, \boldsymbol{\sigma}^{t+\Delta t}) \Delta t \right), \tag{2}$$

In Eq. (2), $\boldsymbol{\sigma}(\mathbf{x})$ is the Cauchy stress tensor, $\mathbf{C}(\mathbf{x})$ is the elastic stiffness tensor, and $\boldsymbol{\varepsilon}(\mathbf{x})$, $\boldsymbol{\varepsilon}^e(\mathbf{x})$, and $\boldsymbol{\varepsilon}^p(\mathbf{x})$ are the total, elastic, and plastic strain tensors, respectively. Using Eq. (2), the total strain can be defined as

$$\boldsymbol{\varepsilon}^{t+\Delta t}(\mathbf{x}) = \mathbf{C}^{-1}(\mathbf{x}) : \boldsymbol{\sigma}^{t+\Delta t}(\mathbf{x}) + \boldsymbol{\varepsilon}^{p,t}(\mathbf{x}) + \dot{\boldsymbol{\varepsilon}}^{p,t+\Delta t}(\mathbf{x}, \boldsymbol{\sigma}^{t+\Delta t}) \Delta t. \tag{3}$$

2.1. Green's approach as a full field solver

Using the elastic strain–stress constitutive relation we can write

$$\sigma_{ij}(\mathbf{x}) = \sigma_{ij}(\mathbf{x}) + C^0{}_{ijkl}\, u_{k,l}(\mathbf{x}) - C^0{}_{ijkl}\, u_{k,l}(\mathbf{x}), \tag{4}$$

where $u_{k,l}(\mathbf{x})$ represents the displacement gradient tensor. With a slight modification of Eq. (4) we obtain

$$\sigma_{ij}(\mathbf{x}) = C^0{}_{ijkl}\, u_{k,l}(\mathbf{x}) + \phi_{ij}(\mathbf{x}), \tag{5}$$

where the term $\phi_{ij}(\mathbf{x})$ is called polarization field represented as

$$\phi_{ij}(\mathbf{x}) = \sigma_{ij}(\mathbf{x}) - C^0{}_{ijkl}\, u_{k,l}(\mathbf{x}). \tag{6}$$

Using the equilibrium equation $\sigma_{ij,j}(\mathbf{x}) = 0$ in combination with Eq. (5) results in the following equation:

$$C^0{}_{ijkl}\, u_{k,lj}(\mathbf{x}) + \phi_{ij,j}(\mathbf{x}) = 0. \tag{7}$$

Using Green's approach for solving the PDE (Partial differential equation) [68] by introducing Green's function $G_{km}(\mathbf{x})$ correlated with the displacement $u_k(\mathbf{x})$ we can write

$$C^0{}_{ijkl}\, G_{km,lj}(\mathbf{x} - \mathbf{x}') + \delta_{im}\delta(\mathbf{x} - \mathbf{x}') = 0. \tag{8}$$

With applying the convolution theorem [69] we obtain

$$\tilde{u}_{k,l}(\mathbf{x}) = \int_{R^3} G_{ki,jl}(\mathbf{x} - \mathbf{x}')\phi_{ij}(\mathbf{x}')d\mathbf{x}'. \tag{9}$$

Solving for Eq. (9) in Fourier space and then performing the inverse transform to get back to the real space, we can represent the strain by the following equation

$$\varepsilon_{ij}(\mathbf{x}) = E_{ij} + FT^{-1}\left(sym\left(\hat{\Gamma}^0_{ijkl}(\mathbf{k})\right)\hat{\phi}_{kl}(\mathbf{k})\right), \tag{10}$$

where the symbols "$\wedge$" and FT^{-1} indicate direct and inverse Fourier transforms, respectively, and $\mathbf{k}$ is a point (frequency) in Fourier Space. The fourth order tensor $\hat{\Gamma}^0_{ijkl}(\mathbf{k})$ is written as

$$\hat{\Gamma}^0_{ijkl}(\mathbf{k}) = -k_j k_l \hat{G}_{ik}(\mathbf{k}); \quad \hat{G}_{ik}(\mathbf{k}) = [C_{kjil}k_l k_j]^{-1}. \tag{11}$$

2.2. Newton Raphson solver for single crystal stress

Solution of Eq. (7) necessitates an iterative procedure to obtain the stress and strain field. If we consider ${\lambda_{ij}}^{(i)}$ and ${e_{ij}}^{(i)}$ to be the initial guess for the strain and stress fields, respectively, using Eq. (6) we have

$${\phi_{ij}}^{(i)}(\mathbf{x}) = {\lambda_{ij}}^{(i)}(\mathbf{x}) - {C^0}_{ijkl}{e_{kl}}^{(i)}(\mathbf{x}), \tag{12}$$

The next guess for the strain field is obtained using Eq. (10)

$$e_{ij}^{(i+1)}(\mathbf{x}) = E_{ij} + FT^{-1}\left(sym\left(\hat{\Gamma}^0_{ijkl}(\mathbf{k})\right)\hat{\phi}^{(i)}_{kl}(\mathbf{k})\right). \tag{13}$$

In order to use directly the stress field rather than the polarization field, Eq. (13) can be written as [70]

$$e_{ij}^{(i+1)}(\mathbf{x}) = E_{ij} + FT^{-1}(\hat{\hat{e}}_{ij}^{(i)} + sym(\hat{\Gamma}^0{}_{ijkl}(\mathbf{k}))\hat{\lambda}^{(i)}_{kl}(\mathbf{k})). \tag{14}$$

An augmented lagrangian scheme is used [71] to nullify the residual R, as a function of the stress tensor $\boldsymbol{\sigma}^{(i+1)}$ and strain tensor $\boldsymbol{\varepsilon}^{(i+1)}$

$$R_k(\boldsymbol{\sigma}^{(i+1)}) = {\sigma_k}^{(i+1)} + {C_{kl}}^0{\varepsilon_l}^{(i+1)}(\boldsymbol{\sigma}^{(i+1)}) - {\lambda_k}^{(i)} - {C_{kl}}^0 e_l^{(i+1)} = 0, \tag{15}$$

In above equation the Voigt notation has been used to express the symmetric tensors σ_{ij} and C_{ijkl} as vectors of size 6

$$\begin{aligned} &\sigma_{ij} \rightarrow \sigma_k, \quad k = 1, 6 \\ &C_{ijkl} \rightarrow C_{kl} \quad k, l = 1, 6. \end{aligned} \tag{16}$$

Solving Eq. (15) using the NR solver gives us

$${\sigma_k}^{(i+1,j+1)} = {\sigma_k}^{(i+1,j)} - \left(\frac{\partial R_k}{\partial \sigma_l}\Big|_{\boldsymbol{\sigma}^{(i+1,j)}}\right)^{-1} R_l(\boldsymbol{\sigma}^{(i+1,j)}), \tag{17}$$

where ${\sigma_k}^{(i+1,j+1)}$ is referred to as the $(j+1)$-guess for the stress field $\sigma_k^{(i+1)}$. Note that "j" enumerates the NR iteration for stress. Coupling Eqs. (3) and (17) results in finding the Jacobian (i.e. Strain derivative relative to the stress)

$$\frac{\partial R_k}{\partial \sigma_l}\Big|_{\boldsymbol{\sigma}^{(i+1,j)}} = \delta_{kl} + {C_{kq}}^0 {C_{ql}}^{-1} + \Delta t \quad {C_{kq}}^0 \frac{\partial \dot{\varepsilon}_q^p}{\partial \sigma_l}\Big|_{\boldsymbol{\sigma}^{(i+1,j)}}. \tag{18}$$

The critical resolved shear stress ${\tau_c}^s(\mathbf{x})$ used in Eq. (1) is itself a function of stress ${\tau_c}^s\left(\boldsymbol{\sigma}^{(i+1,j)}(\mathbf{x})\right)$. The Jacobian term $\frac{\partial \dot{\varepsilon}_q^p}{\partial \sigma_l}\Big|_{\boldsymbol{\sigma}^{(i+1,j)}}$ in Eq. (18) therefore can be defined as

$$\frac{\partial \dot{\varepsilon}_q^p}{\partial \sigma_l}\Big|_{\boldsymbol{\sigma}^{(i+1,j)}} \cong n\dot{\gamma}_0 \sum_{s=1}^{N} \frac{{m^s}_q {m^s}_l}{{\tau_c}^s\left(\boldsymbol{\sigma}^{(i+1,j)}(\mathbf{x})\right)} \left(\frac{\mathbf{m^s}(\mathbf{x}):\boldsymbol{\sigma}(\mathbf{x})}{{\tau_c}^s\left(\boldsymbol{\sigma}^{(i+1,j)}(\mathbf{x})\right)}\right)^{n-1}. \tag{19}$$

Incorporating Eq. (19) into Eq. (18) gives

$$\frac{\partial R_k}{\partial \sigma_l}\Big|_{\boldsymbol{\sigma}^{(i+1,j)}} \cong \delta_{kl} + {C_{kq}}^0 {C_{ql}}^{-1} + (\Delta t\, n\, \dot{\gamma}_0){C_{kq}}^0 \sum_{s=1}^{N} \frac{{m^s}_q {m^s}_l}{{\tau_c}^s\left(\boldsymbol{\sigma}^{(i+1,j)}(\mathbf{x})\right)} \times \left(\frac{\mathbf{m^s}(\mathbf{x}):\boldsymbol{\sigma}(\mathbf{x})}{{\tau_c}^s\left(\boldsymbol{\sigma}^{(i+1,j)}(\mathbf{x})\right)}\right)^{n-1}. \tag{20}$$

By minimizing the residual R_k over the iterative NR solver, the stress solution for each FFT sampling point is obtained as the next guess for Eqs. (12) and (14). The procedure is continued until convergence is achieved by reaching a pre-defined tolerance (TOL_{NR}) for single crystal stress defined as

$$\frac{\left({\sigma_k}^{(i+1,j+1)} - {\sigma_k}^{(i+1,j)}\right)\left({\sigma_k}^{(i+1,j+1)} - {\sigma_k}^{(i+1,j)}\right)}{\sqrt{{\lambda_{ij}}^{(i)}{\lambda_{ij}}^{(i)}}} \prec TOL_{NR} = 10^{-7} \tag{21}$$

The stress and strain field tolerance (TOL_{stress_field} , TOL_{strain_field}) after solving Eq. (15) can be expressed according to Eqs. (22a) and (22b)

$$\frac{\left\langle\left({\sigma_k}^{(i+1)} - {\lambda_k}^{(i)}\right)\left({\sigma_k}^{(i+1)} - {\lambda_k}^{(i)}\right)\right\rangle}{\sqrt{\frac{3}{2}{\Sigma'_{ij}}^{(i)}{\Sigma'_{ij}}^{(i)}}} \prec TOL_{stress\_field} = 10^{-6} \tag{22a}$$

$$\frac{\left\langle\left({\varepsilon_k}^{(i+1)} - {e_k}^{(i)}\right)\left({\varepsilon_k}^{(i+1)} - {e_k}^{(i)}\right)\right\rangle}{\sqrt{\frac{2}{3}{E_{ij}}^{(i)}{E_{ij}}^{(i)}}} \prec TOL_{strain\_field} = 10^{-6} \tag{22b}$$

where, Σ'_{ij} and E_{ij} denote the macro deviatoric stress and strain response of the whole polycrystal by taking the average of the fields over the RVE domain.

3. Simple compression and plain strain compression of oxygen free high conductivity (OFHC) copper

In order to compare the accuracy of the developed parallel implementations, we performed a plain strain compression (PSC) case study, in which the deformation behavior and texture evolution of an oxygen free high conductivity (OFHC) polycrystalline copper are simulated. The sample RVE underwent PSC with applied strain rate of 0.001 (1/s) up to the accumulated equivalent strain level of 0.5. The copper polycrystal is composed of face-centered cubic (FCC) grains with random orientation distribution. Crystals are represented using VORONI Tessellation, in which, the unit cell of polycrystalline is partitioned into subdomains or grains [72,73]. The simulations are performed for different number of FFT sampling points of 16^3 (4096 voxels), 32^3 (32,768 voxels), 64^3 (262,144), and 128^3 (2,097,152). The elastic material properties for OFHC copper at room temperature were set to be $C_{11} = 170.2$ (GPa), $C_{12} = 114.9$ (GPa) and $C_{44} = 61.0$ (GPa) [74]. A nonlinear voce type hardening law was used to generate the stress–strain curves and can be expressed as follows [75]

$${\tau_c}^s = {\tau_0}^s + ({\tau_1}^s + {\theta_1}^s\Gamma)\left(1 - \exp\left(-\Gamma\left|\frac{{\theta_0}^s}{{\tau_1}^s}\right|\right)\right), \tag{23}$$

where, $\Gamma = \sum_s \Delta\gamma^s$ is the accumulated shear in the grain; ${\tau_0}^s$, ${\theta_0}^s$, ${\theta_1}^s$, $({\tau_0}^s + {\tau_1}^s)$ are the initial critical resolve shear stress (CRSS), initial hardening rate, the asymptotic hardening rate, and the back-extrapolated CRSS. The values used for the hardening parameters in our simulations were calibrated to be ${\tau_0}^s = 14.5$ MPa, ${\tau_1}^s = 99$ MPa, ${\theta_0}^s = 250$ MPa, ${\theta_1}^s = 14$ MPa. Another type of hardening is a dislocation density (DD) hardening law which is more involved than the Voce type. Readers are referred to [76–78] for detailed information about DD hardening.

Fig. 1 represents the true stress–true strain curve for this simulation for both serial, MPI, and GPU versions of the EVPFFT. Results show that, as expected, both serial and parallel solvers are on top of each other, and the calculation closely resembles the measured stress. Further validation is provided by evaluating the evolved texture (i.e. evolution of crystal orientations relative to the sample axis) and the stress and strain fields represented in Fig. 2, where the contours of equivalent plastic strain and von Mises stress are shown over the RVE at the accumulated strain of 0.5. The fields for both serial and parallel versions of the code are indistinguishable.

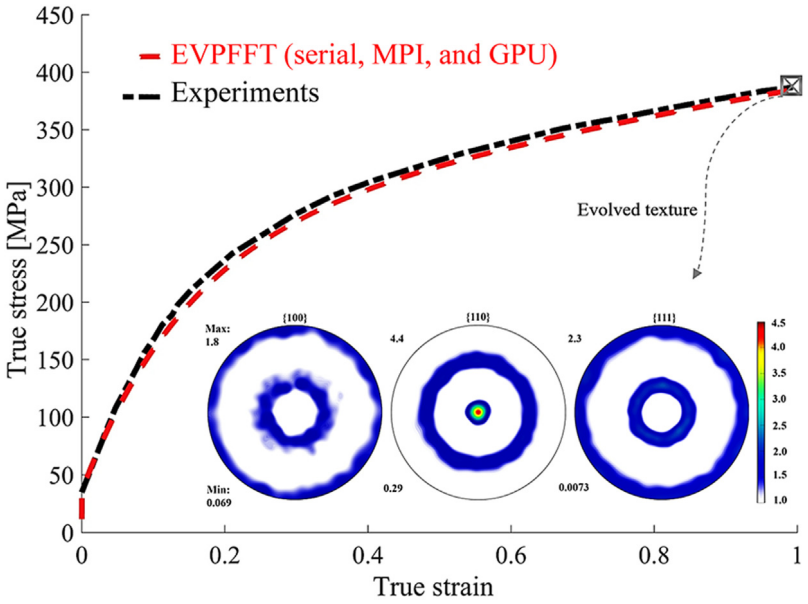

Fig. 1. True stress–strain curves calculated using the original serial EVPFFT and parallel versions of the code for simple compression of oxygen free high conductivity (OFHC) copper (Cu) to a strain of 1.0. The experimentally measure curve is included to compare the quality of the fit.

4. EVPFFT intensive computations — the hotspot analysis

The EVPFFT code was profiled using the Portland Group Inc. (PGI) performance profiler 2018 v18.30 [79]. This was done for four different problem sizes of 16^3, 32^3, 64^3, and 128^3 representing the total number of FFT points (voxels) in the representative volume element (RVE). Fig. 3 represents the distribution of hotspots (i.e. intensive computations) throughout different parts of the code. Evidently, the NR iterative solver for Eq. (17) including the elasto-plastic decomposition and the Jacobian calculation, accounts for up to 85% of the code for all problem sizes. With increase in problem size, the FFT function becomes more computationally expensive taking a higher percentage of the total CPU time.

In high performance computing, the objective is to focus on accelerating the dominant routines (i.e. most time consuming portions). That is because according to Amdahl's Law, there is little benefit in speeding up a small portion of a code (e.g. less than 10%), because even if infinitely accelerated, the total performance gain will be small and generally not worth the effort. The NR (i.e. Eqs. (17)–(20)) hotspot accounts for the largest portion of the processing time, making it the most crucial part to parallelize.

In the next section, we elaborate on using OpenACC to port the NR solver to run in parallel on GPU. The environment used for this purpose is provided in Table 1. Note that NVIDIA PGI compiler 2018 v18.30 is used to generate all the data presented in this work. GPU architectures of Kepler (K80 and k20X), Pascal (P100) and Volta (V100) used in this research are compared in detail in Table 2.

5. Background on GPU, OpenACC, and CUDA

OpenACC, originally developed by major vendors CAPS [80], CRAY [81], and NVIDIA PGI [82], is a high level performance-portable parallel programming model based on directives/pragmas to enable scientists and programmers to accelerate their codes without changing the code structure significantly [64]. The main reason behind using OpenACC is to maintain performance portability of a given code. In some cases, OpenACC can result in a better efficiency compared to its peers CUDA and OPENCL [83,84] due to the high level of available optimizations provided with the OpenACC compiler.

OpenACC directives tell the compiler to translate the OpenACC region/loops into kernels that run either on CPU or GPU. In the NVIDIA PGI compiler for instance, passing "-ta=tesla" utilizes the GPU while "-ta=multicore" utilizes the CPU cores for a multithreaded execution. In case of GPU utilization data is copied from CPU (also called "host" in OpenACC) local memory to the GPU (also called "device") dedicated memory and then the calculations are performed on the GPU. The GPU programming model employs a grid of blocks of threads. Both OpenACC and CUDA will map loop statements to kernels launching on blocks of threads [85].

5.1. ACC routines

In order to call a routine or function from the device code, all of the routines should include the OpenACC directive "!$ACC routine X", in which, "X" is chosen to be "seq", "vector", "worker" or "gang". The "!$ACC routine seq" tells the compiler to put a serial copy of the routine on the GPU, while the two later ones imply thread and block parallelism level of the routine on the device. The most common practice for inner loops and routines inside a device code is to use the "!$ACC routine seq", however, in order to improve the efficiency, the inner loops should be optimized to run on groups of thread blocks.

5.2. Tuning the performance using number of threads and thread blocks

The NVIDIA GPU architecture incorporates several streaming multiprocessors (SM). When an OpenACC or CUDA program runs, grid blocks are distributed to multiprocessors. Thread blocks are executed by SM that are designed to execute several threads concurrently. Threads are executed in groups of 32 that are called a warp. 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", "worker" and "vector" features are used to control the total number of thread blocks and threads per block used

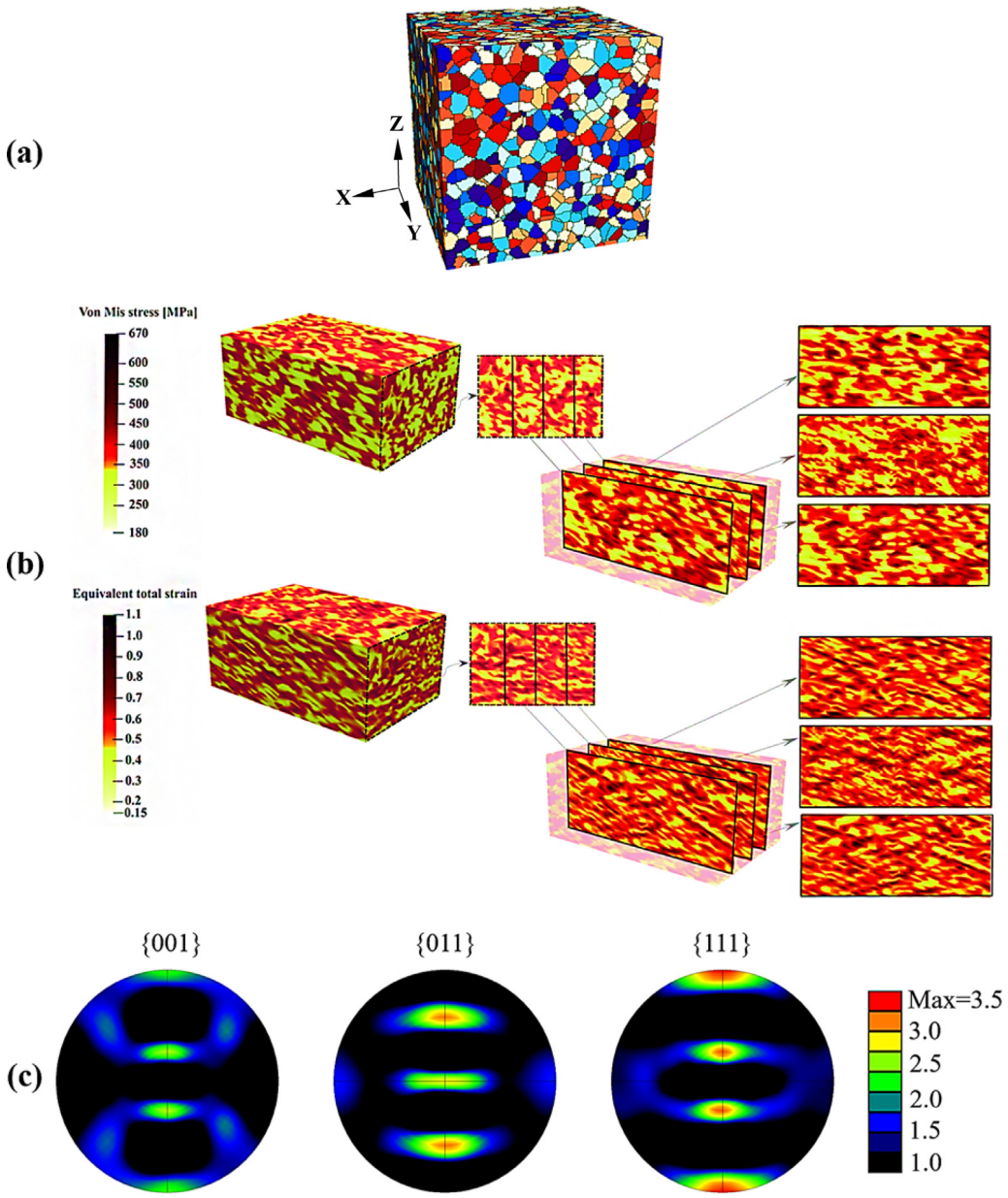

Fig. 2. (a) Initial microstructure of polycrystalline RVE. (b) Von Mises stress and equivalent total strain fields simulated under plain strain compression at an accumulated strain 0.5 of OFHC. Cutting planes at different locations of RVE illustrate different response in field contours due to the microstructure anisotropy (i.e. heterogeneity of grain orientation distribution). (c) Texture evolution of OFHC copper under plain strain compression measured at accumulated strain of 0.5.

Table 1
Hardware specs of the workstation.

Compiler	OS	CPU	Kepler GPU (s)	Pascal GPU (s)	CUDA toolkit version	System memory (GB)	# of CPUs	# of threads per core	# of sockets	# of cores per socket
NVIDIA PGI v 2017 v17.5	Centos 7.0	Intel(R) Xeon(R) CPU E5-2695 v4 @ 2.10 GHz	NVIDIA Tesla K80 (2 GPUs)	NVIDIA Tesla P100	9.1	512	36	2	2	18

Table 2
NVIDIA Tesla K20x and K80 (Kepler) vs. Tesla P100 (Pascal) and Tesla V100 (Volta): hardware specs and performance.

GPU	Architecture	Streaming multiprocessors (SM)	CUDA cores	Frequency (MHz)	GFLOPS (double)	Memory (GB)	Memory bandwidth (GB/s)
K20X	Kepler	14	2688	732	1310	6	250
K80	Kepler	2 × 13	2 × 2496	562	2 × 1455	2 × 12	2 × 240
P100	Pascal	56	3584	1126	4670	16	720
V100	Volta	84 (80 usable)	5120	1355	7800	16	900

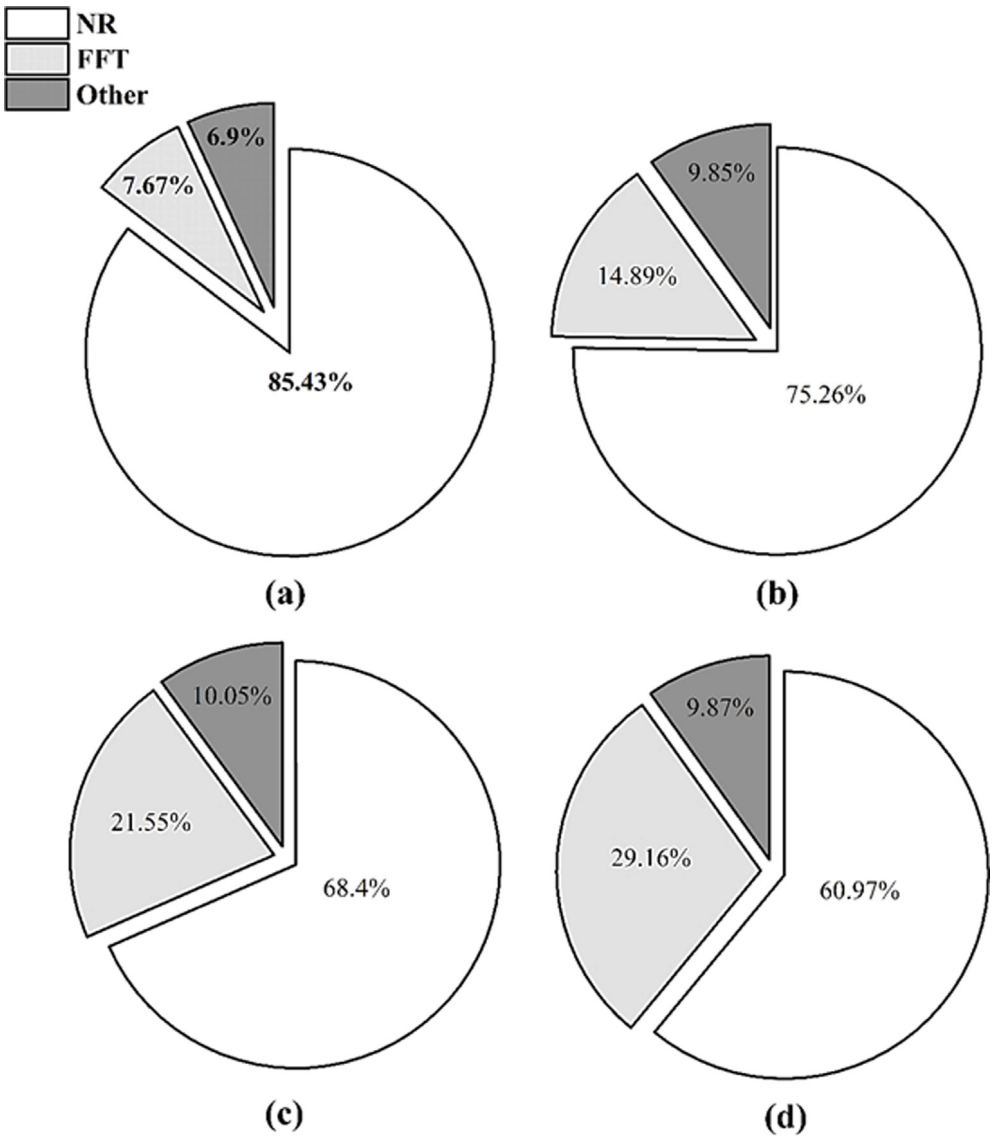

Fig. 3. Workload distribution within EVPFFT solver for RVE domain sizes of: (a) 16^3, (b) 32^3, (c) 64^3, and (d) 128^3.

for running CUDA kernels. For the GPU architectures invented so far, the NVIDIA PGI compiler maps "gang" with thread-blocks ("gridDim"), "workers" with threads "blockDim.y", and "vector" with threads "blockdim.x". Note that this is currently the type of binding observed by using the compiler flag "-Minfo=accel". This type of binding is prone to possible changes for future generation of GPUs.

While the "!$ACC kernels" and "!$ACC parallel" both perform the same task (i.e. compile the code to launch on GPU), they do not necessarily result in identical compiler outcome. When the "!$ACC kernels" is used, the compiler decides if it is safe (i.e. ensure correctness of parallel run on GPU compared to run on CPU) to parallelize a loop. In other words, if the compiler decides (i.e. whether right or wrong) there is any potential dependency of the data inside the loop parallelism is going to be avoided. In contrast, the "!$ACC parallel" is used to enforce the compiler what the developer intends to do. Therefore, using the later one demands a careful implementation and to ensure no data dependencies exist (e.g. the value of an array in the next iteration does not depend on its value obtained from the previous iteration) before moving forward, otherwise, it may result in wrong results.

In order to map the loops on blocks and threads, depending on whether "!$ACC kernels" or "!$ACC parallel" is used, the "!$ACC gang(gang_size)" and the "!$ACC vector(vector_size)" are used for the "!$ACC kernels" implementation, while the "!$ACC num_gangs(gang_size)" and the "!$ACC vector_length (vector_size)" are utilized for the "!$ACC parallel" implementation. In our code, we use the "!$ACC parallel" and the correspond directive for mapping the loops into threads and thread blocks. Like mentioned before, the "!$ACC parallel" gives us full control over the code allowing us to determine which loops need to be run sequentially, on multiple threads, on thread blocks or on a combination of both (i.e. running on both thread blocks and threads). We managed to tune the code using the "!$ACC parallel" together with "!$ACC num_gangs" and "!$ACC vector_length" by running the inner loops with higher compute intensities on multiple threads instead of running them sequentially on the GPU.

5.3. OpenACC-CUDA interoperability

When CUDA libraries (e.g. CUFFT, CUBLAS, …) [65,86] are called from the host (i.e. CPU) code, the data transfers from GPU memory to CPU memory (and vice versa) are performed using the CUDA built-in functions (i.e. CudaMemcpy()). Our

```
!$ACC data create(…) present(…)
!$ACC update device(…)
!$ACC parallel num_gangs(…) vector_length(…)
!$ACC loop independent gang vector collapse(3)
!$ACC& private(…)
!$ACC& reduction(+:…)
Do i=1, Number of voxels in X direction (e.g. 16,32,64,128)
  Do j=1, Number of voxels in Y direction (e.g. 16,32,64,128)
    Do k=1, Number of voxels in Z direction (e.g. 16,32,64,128)
      [Write global data of all voxels to local data per voxel]
       Do while convergence met for single crystal stress
        [NR solver per voxel]
       enddo
    enddo
  enddo
enddo
!$ACC end data
```

Fig. 4. Pseudo code for GPU NR single crystal stress solver. The actual NR iteration loop (i.e. Do While loop) is called for all RVE voxels through the three tightly nested outermost loops. The OpenACC kernel directives tell the compiler to run the NR solver on the device (i.e. GPU) rather than host (i.e. CPU). The ACC data directives control the data transfer (i.e. Memcopy) between CPU and GPU. In our final implementation, most of the data is already present on the GPU, removing the need for copying back and forth between CPU and GPU memories.

goal is to maintain performance portability as much as possible. With the advent of OpenACC-CUDA interoperability [87], calling CUDA functions in OpenACC regions is viable. It is also worth mentioning that using OpenACC-CUDA interoperability, we are able to use OpenACC directives for CPU–GPU data transfers more conveniently and consistently specially when using unstructured OpenACC data regions [84], in which, "ACC enter/exit data" as heterogeneous data structures are used to keep track of data copy among different routines running on GPU [84].

6. Porting NR to GPU using OpenACC

The NR solver was ported to the GPU using OpenACC directives. Fig. 4 describes a Pseudo code for NR representing the GPU implementation using OpenACC parallel and data directives. Readers can refer to [84] for detailed information about OpenACC and how to employ it efficiently for porting a code to run on a GPU-based hardware.

Note that the NR solver includes several subroutine calls inside the three nested loops iterating over all the voxels. Calling subroutines from device code, necessitates a GPU copy of the routine to run on the device.

Supplementary material presents the NR subroutine on GPU illustrating where the internal loops are located and targeted for "!$ACC routine". Indeed, calculation of shear stress, Jacobian, and tensor rank conversions (i.e. 6×6 Voigt notation to $3 \times 3 \times 3 \times 3$ and vice versa) are the major computations for these inner loops. In order to ensure maximum performance, a combination of "!$ACC routine seq" and "!$ACC routine vector" was used. The former one is aimed for small loops with few iterations, since, it is not efficient to run few iterations parallel on a GPU.

6.1. Matrix inversion routine in NR: LU vs. GJE

6.1.1. LU decomposition and its drawback for NR device code

Calculation of $(\frac{\partial R_k}{\partial \sigma_l}\big|_{\boldsymbol{\sigma}^{(i+1,j)}})^{-1}$ in Eq. (17), which is the inversion of the Jacobian matrix, is the hotspot in the NR solver. The original inversion routine used in EVPFFT is the LU decomposition, which due to data dependency runs sequentially on each thread. Here, each thread performs one LU inversion for its respective voxel while the LU itself is sequential. Therefore, we investigate alternatives in order to find a more efficient matrix inversion method for the NR solver running on the GPU. In order to avoid using ACC sequential routine for the LU inversion being called from the device code, an alternative matrix inversion algorithm of Gauss Jordan Elimination (GJE) is explored. In the next section we elaborate on performance benchmark of such inversion method and compare to the traditional LU.

6.1.2. Gauss Jordan elimination (GJE)

An alternative to the LU decomposition is GJE algorithm [88–92], which is known as efficient for smaller matrix dimensions comparing to its former version named Gaussian Elimination. The fact that GJE is faster than its alternatives for small matrix dimensions, makes it a good candidate for our NR code in which most of arrays have a size of 3×3 (e.g. stress and strain tensors) and 6×6 (e.g. Jacobian tensor) per voxel. In order to evaluate the performance of GJE over the LU inversion, we performed a benchmark in which square matrixes with dimensions of 3×3, 6×6 are inverted repeatedly over voxels of RVE resolutions of 16^3, 32^3, 64^3, and 128^3. The reason behind this is to mimic what happens in NR solver structure (see Fig. 4.) Fig. 5 represents the performance benchmark for GJE and LU. We find that for matrix dimensions of 3×3 and 6×6 which are frequently used in NR, the GJE inversion performs several times faster than the LU method for the RVE size of 128^3, running standalone (i.e. outside of EVPFFT solver) on a single CPU core.

Next, we incorporate the GJE into the NR subroutine to evaluate how it impacts the performance of the overall code. The call to GJE is from the device since the NR is running on GPU using OpenACC. We found that replacing the LU inversion with GJE can improve the NR performance by 42% for the RVE size of 128^3. After incorporating GJE, the next step is to further tune the GPU NR by further loop optimizations.

6.2. Loop collapse optimization

In most cases, merging the iterations of the tightly nested loops into a single united large loop providing more iterations, results in performance improvement for any kind of parallel implementation whether on CPU cores or GPU (i.e. OpenMP/OpenACC). This is due to the higher degree of parallelism and potentially better balancing of the work distributed over threads. Note that we apply this optimization to all nested loops within NR to ensure

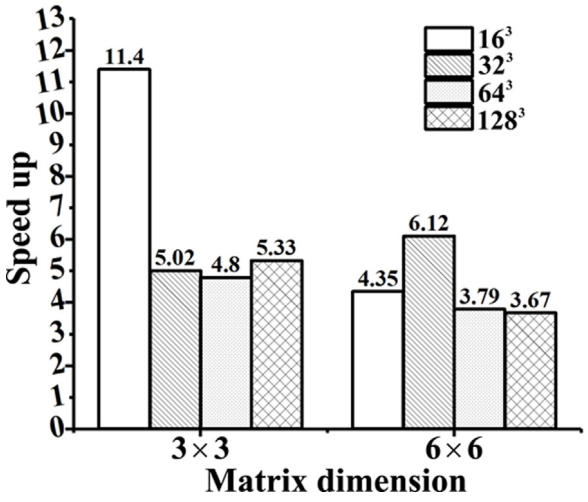

Fig. 5. Speedup of the GJE matrix inversion algorithm over the LU running standalone on single CPU for different RVE sizes of 16^3, 32^3, 64^3, and 128^3 and square matrix sizes of 3×3 and 6×6.

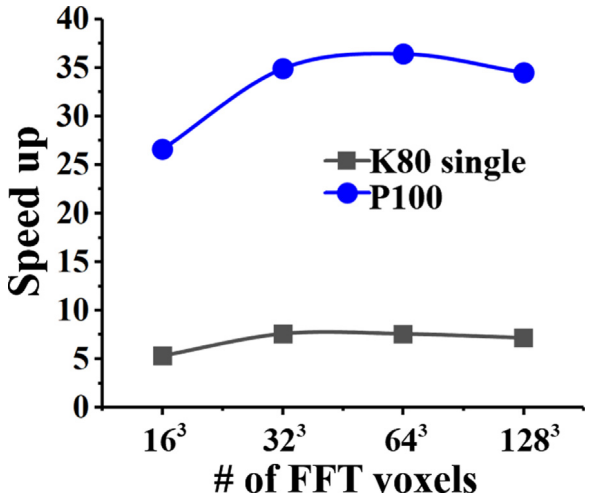

Fig. 6. Speedup of GPU NR solver in EVPFFT code running on NVIDIA Tesla K80 (single) and NVIDIA Tesla P100 devices over the scalar version for RVE domain sizes (i.e. voxel resolution) of 16^3, 32^3, 64^3, and 128^3. As is evident, the P100 GPU outperforms the K80 significantly due to its high performance Pascal architecture.

maximum performance. We find that collapsing loops results in up to 91% NR kernel speedup (for RVE resolution of 128^3) compared to the case that only the outermost loop is parallelized.

6.3. Tuned NR scalability running on GPU

Fig. 6 represents the final NR speedup obtained running on P100 and K80 (single) NVIDIA Tesla GPUs after incorporating all improvements and optimizations discussed earlier in the text. The performance on NVIDIA Tesla P100 is significantly higher than K80 due to its innovative Pascal architecture [93], speeding up the NR solver for 36.42x over the serial version. As a next step, we compare the performance of P100 with the multicore CPU implementation of the NR subroutine in the EVPFFT code [56] on up to 64 MPI processes on Intel Xeon 2695 v4. Results of this benchmark are summarized in Fig. 7. We find that P100 outperforms the MPI multicore version for all RVE resolutions running on any number of MPI processes up to 64.

A total speedup achieved by the ACC-EVPFFT implementation gained by accelerating the NR routine is possible to estimate using Amdahl's law. This is illustrated in Fig. 8, where the net performance of EVPFFT is shown after accelerating the NR subroutine only. Even if we infinitely speedup the NR solver, a maximum net gain for the whole code is not more than 6.86x, 4.04x, 3.17x, and 2.56x for RVE resolutions of 16^3, 32^3, 64^3, and 128^3, respectively. Such a trend is largely due to a growing importance of the FFT calculations, as shown in Fig. 3. In the next section, we focus on accelerating the FFT calculations on GPUs using the CUDA libraries.

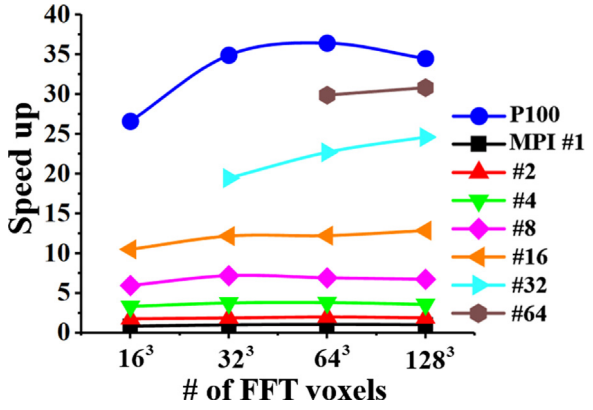

Fig. 7. Performance benchmark of GPU NR solver in EVPFFT code running on NVIDIA Tesla P100 device vs. MPI multicore CPU version running on up to 64 MPI processes on Intel Xeon 2695 v4 for RVE domain sizes (i.e. voxel resolution) of 16^3, 32^3, 64^3, and 128^3. The NVIDIA Tesla P100 outperforms the MPI multicore version for all RVE resolutions running on any number of MPI processes up to 64.

7. FFT libraries: GPU vs. multicore CPU

One of the common FFT solvers used in a wide variety of scientific codes, including the original EVPFFT solver, is the "FOURN" routine, presented in Numerical Recipes in FORTRAN and C++ [94,95]. Although this routine is commonly used, more advanced libraries have been recently developed to perform FFTs with a higher level of efficiency. FFTW and its MPI version [57,96–99] and CUFFT [65,100,101] are currently among the fastest FFT libraries, running on a single CPU core, multiple CPU cores (i.e. for the MPI version of FFTW3), and GPUs, respectively. FFTW is an efficient FFT library developed at Massachusetts Institute of Technology (MIT) for computation of discrete Fourier transforms (DFT) in 1D, 2D, and 3D space [97]. FFTW supports both real and complex data input with arbitrary size transformations. In the present work, we have used FFTW 3.3.6, which is the latest release of the FFTW library. FFTW3 supports all of the SSE (i.e. Streaming SIMD (Single Instruction Multiple Data) Extensions)/SSE2/AVX (i.e. Advanced Vector Extensions)/ARM (i.e. Advanced RISC (Reduced Instruction Set Computer) Machines) instructions for CPU hardware optimizations [102].

On the other hand, CUFFT (i.e. CUDA FFT) is the NVIDIA CUDA FFT library developed after FFTW and is able to speedup the FFT computations drastically running on the thousand CUDA cores. However, the CUFFT performance may vary depending on the GPU hardware architecture and specifications (i.e. CUDA cores, memory speed, etc.). CUFFT supports similar capabilities to FFTW and large data sets of up to 128 million elements. The CUFFT library is accessible as a part of the NVIDIA CUDA toolkit [103]. In what follows, we discuss and compare the performance of CUFFT vs. FFTW library.

In order to benchmark the performance of CUFFT and FFTW libraries, a study, comparing standalone FFTs with a complex

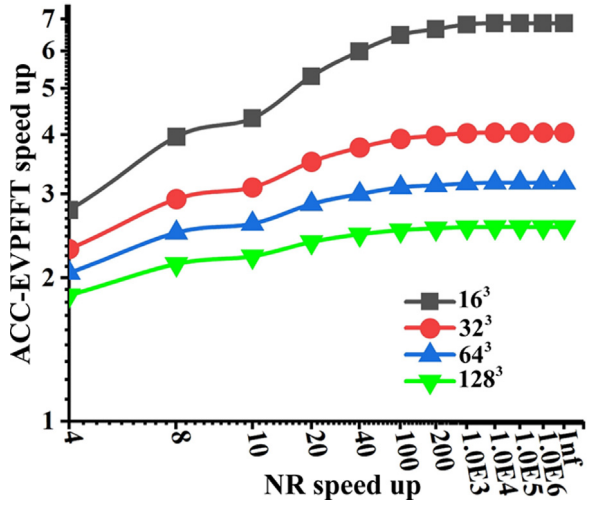

Fig. 8. Theoretical ACC-EVPFFT speedup as a function of NR speedup generated by Amdahl's law for RVE domain sizes of 16^3, 32^3, 64^3, and 128^3. Speeding up the NR infinitely, maximum net gain for the whole code is up to 6.86x, 4.04x, 3.17x, and 2.56x for RVE resolutions of 16^3, 32^3, 64^3, and 128^3, respectively. This is due to dominant computational expense in FFT calculations running on larger RVE sizes as indicated in Fig. 3.

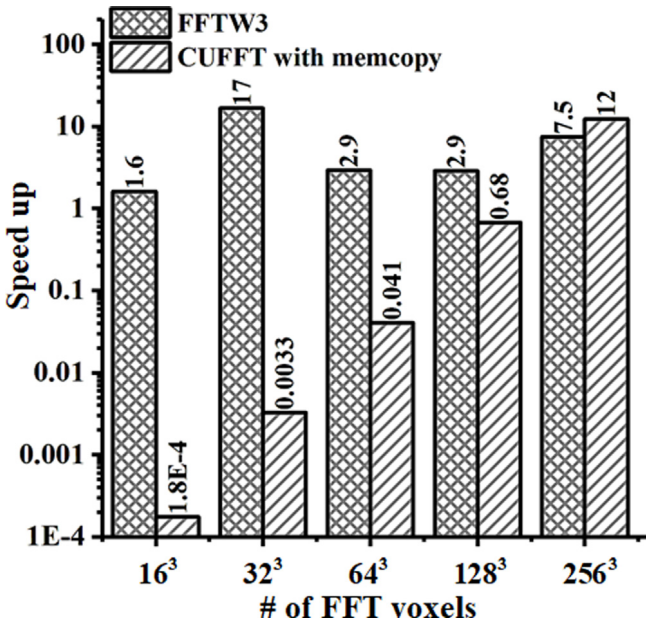

Fig. 9. Speedup of standalone FFTW3 and CUFFT running on single core of Intel Xeon 2695 v4 and single Tesla K80, respectively, over the FOURN for different RVE resolutions (voxel points). The CUFFT run includes inevitable memcopy operations from CPU to GPU and vice versa which is accounted in the results presented in the figure.

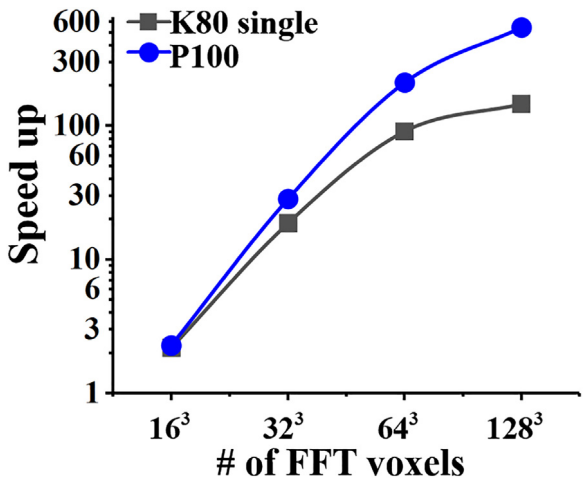

Fig. 10. CUFFT scalability running on NVIDIA Tesla K80 (single) and P100 relative to FOURN for RVE domain sizes of 16^3, 32^3, 64^3, and 128^3. The single K80 and P100 NVIDIA GPUs perform 145.7x and 545.6x, respectively, faster than the original FOURN subroutine. The Pascal architecture outperforms the Kepler hardware 3.74x for the largest domain size of 128^3.

data type input is conducted. Note that the benchmark tests and the following result are specific to our implementations and the hardware specifications we use in this research (Table 1).

Fig. 9 illustrates this benchmark comparing different RVE resolutions in 3D transformations on Intel Xeon 2695 (FFTW3) v4 and single K80 GPU (CUFFT). We find that, starting from the RVE size of 256^3, the CUFFT library outperforms the FFTW. This behavior is justified by the fact that GPU works more efficiently than a CPU for large data sets and massive parallel tasks due to its hardware architecture with thousands of built-in computing cores. It is also worth mentioning that a significant part of a GPU runtime is spent on data transfer (if not possible to avoid) to and from the CPU. Eliminating this data transfer, i.e. making the data always resident on the GPU, should result in a considerable performance gain. Such data management is feasible using OpenACC data regions to keep the data present on GPU as much as possible [84]. We will elaborate on such data interoperability in detail in the next section.

Fig. 10 represents the scalability of CUFFT on K80 (single) and P100 GPUs inside the EVPFFT solver, normalized to the original FOURN running on single Intel Xeon 2695 v4 core. Increasing the total number of FFT points result in better speedup for the GPU case due to more parallelism potential available for GPU threads. We find that single K80 and P100 NVIDIA GPUs perform up to 145.7x and 545.6x, respectively, faster than the original FOURN subroutine. It is also important to observe that the Pascal architecture outperforms the Kepler hardware for 3.74x for the largest domain size of 128^3. We also assess a more rigorous performance benchmark for CUFFT, by a comparison against the MPI version of FFTW3 (MPI-FFTW3), which was reported on in detail in our previous work [56]. The results are provided in Fig. 11. We find that CUFFT outperforms the MPI-FFTW3 drastically starting form RVE size of 32^3 with a growing scalability as a function of RVE size.

8. GPU acceleration of remaining routines

Running NR and FFTs on GPU using OpenACC and CUDA, the main portion of the code is GPU accelerated according to the workload distribution provided early in the text in Fig. 3. However, since other routines of the code are not running on GPU, the data transfer between CPU and GPU back and forth is unavoidable. This becomes more important when the invocations occur frequently due to highly nested iterations of NR inside the FFT equilibrium field iterations. Same analogy fits for the data copy required for the FFT calculations. In order to mitigate this issue, we ported other routines to the GPU, except the I/O subroutines that read and write data. While some data copies are still inevitable, the fraction of data transfer time compared to the kernel compute time is small. Fig. 12 represents the compute (i.e. kernel launch on GPU) and Memcpy (i.e. memory copy between host and device) for the ACC-EVPCUFFT running on the P100 device, provided by NVIDIA PGI performance profiler 2018.

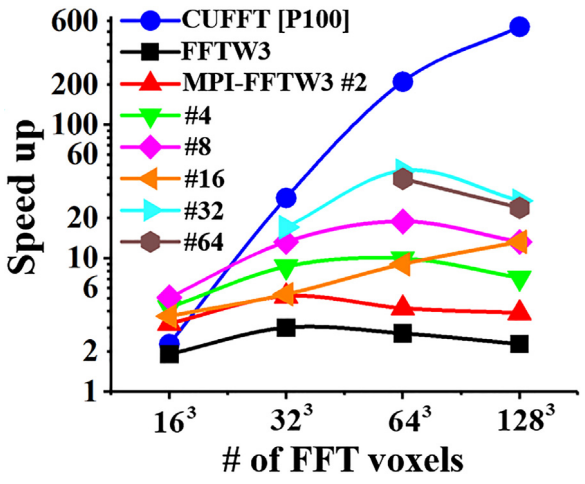

Fig. 11. Performance of CUFFT library running on NVIDIA Tesla P100 compared to the MPI version of FFTW3 library (i.e. MPI-FFTW3) running on 1–64 CPU cores of Intel Xeon 2695 v4 for RVE domain sizes of 16^3, 32^3, 64^3, and 128^3. The data is normalized with the original FOURN FFT. As is evident, CUFFT outperforms the MPI-FFTW3 drastically with a growing scalability as a function of the RVE size.

It is clear from the profiler that GPU is kept busy most of the time to ensure minimum gaps generated by un-accelerated regions (routines running on the host). Note that the remaining small data copies are owing to unavoidable data copies into GPU ("ACC update device") for small local arrays of size 6×6, which take orders of micro seconds and originated from MPI communications owing to their use in the MPI reduction and the overall control flow of parallelization. The major focus on keeping data present on GPU is related to the global large arrays for which the information is stored per voxel. Maximum achievable performance is gained by porting most of the code to the GPU as indicated by the profiler. It is also worth mentioning that the profiler reveals NR still as the major hotspot of the calculations even after porting all to the GPU. This is clear comparing the compute duration of NR relative to FFTs or other kernels launching on GPU.

The final net speedup for ACC-EVPCUFFT after porting all routines to GPU (except those performing read and write) is represented in Fig. 13 running on single K80 and P100 devices normalized over the serial version running on single Intel Xeon 2695 v4 core for RVE domain sizes of 16^3, 32^3, 64^3, and 128^3. As expected, the P100 GPU performs up to 2.94x faster than K80 with a net speedup of up to 27.3x for the RVE size of 128^3. The result of P100 performance benchmark against the MPI version of the code running on up to 64 CPU cores is illustrated in Fig. 14. We find that P100 GPU slightly outperforms running the code with MPI decomposition over 64 Intel Xeon 2695 v4 processes for the RVE size of 128^3. This again has to do with the capability of GPUs in running massive data sets more efficiently comparing to CPUs.

In summary, while single GPU operated by a single CPU performs significantly better than MPI for the NR (see Fig. 7) and FFT (see Fig. 11) calculations, this is not the case for other subroutines. This is due to the fact that those subroutines are less favorable for GPU acceleration due to less arithmetic intensities and being compute bound. However, porting all routines to GPU is vital to ensure minimizing the data transfer between CPU and GPU as indicated in Fig. 12. Nevertheless, P100 GPU still outperforms 64 Intel Xeon 2695 v4 processes for the RVE size of 128^3 and greater.

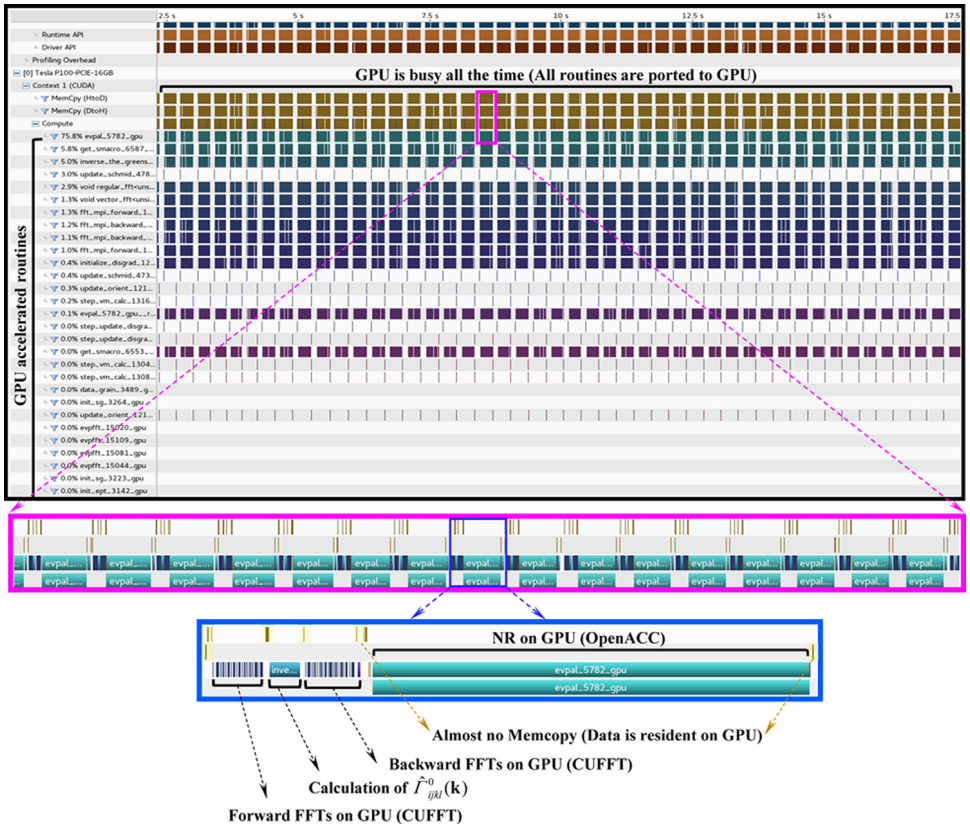

Fig. 12. Distribution of Compute (i.e. kernel launch on GPU) and Memcpy (i.e. memory copy between host and device) for the EVPFFT code running on the P100 device, provided by NVIDIA PGI performance profiler 2018. Profiler indicates that GPU is kept busy most of the time doing computations. This is a result of porting almost all routines to GPU using OpenACC as indicated in the figure.

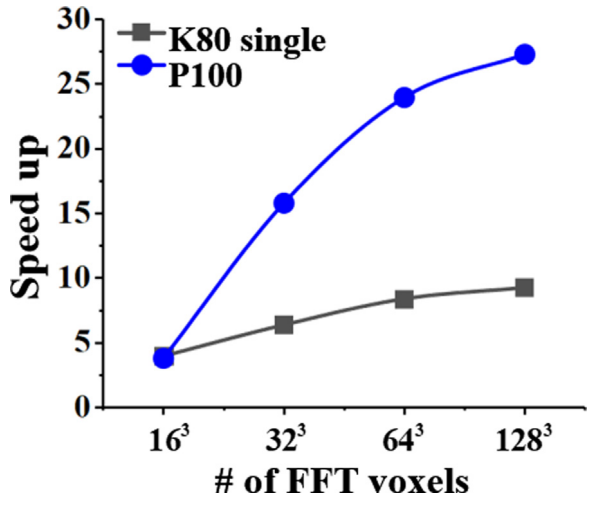

Fig. 13. ACC-EVPCUFFT net scalability running on NVIDIA Tesla K80 (single) and P100 GPUs normalized over the serial version running on single Intel Xeon 2695 v4 core for RVE domain sizes of 16^3, 32^3, 64^3, and 128^3. The P100 GPU performs up to 2.94x faster than K80 with a net speedup of up to 27.3x for the RVE size of 128^3.

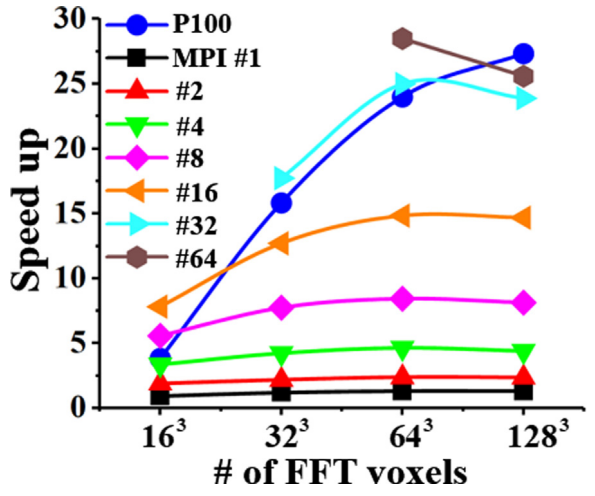

Fig. 14. Performance benchmark of ACC-EVPCUFFT code running on NVIDIA Tesla P100 device vs. MPI multicore CPU version (MPI-EVPFFTW) running on up to 64 MPI processes on Intel Xeon 2695 v4 for RVE domain sizes (i.e. voxel resolution) of 16^3, 32^3, 64^3, and 128^3. The NVIDIA Tesla P100 slightly outperforms running the code with MPI decomposition over 64 Intel Xeon 2695 v4 processes for the RVE size of 128^3 and this is due to the capability of GPUs in running massive data sets more efficiently than CPUs.

It is also worthwhile noting that NVIDIA has released the most recent Tesla architecture, called Volta (i.e. Tesla V100) [104] which is claimed to be up to 3.133x faster than Tesla P100. Table 2 represents the hardware specs and performance of NVIDIA Tesla V100. If the ACC-EVPCUFFT code is run on a single V100, speed ups could potentially improve up to approximately 3x.

9. Multi-GPU implementation combining OpenACC with MPI (MPI-ACC-EVPCUFFT)

In order to run ACC-EVPCUFFT on multiple GPUs, we leverage our previous work that used the domain decomposition approach and the message passing interface (MPI) standard [105–107] to provide capability of utilizing many GPUs in a GPU cluster. To this end, the very outermost loops over the FFT voxels (see Fig. 4) are split into chunks of data (i.e. domain decomposition). Fig. 15 shows a schematic view of this domain decomposition for 4 GPUs in one direction.

In order to enable the code with multi-GPU computation, one needs to detect and switch between different devices any time needed. To this end, the OpenACC built-in functions included in OpenACC library headers are utilized by adding the "USE OpenACC" statement right after the routine declaration. This is important in a FORTRAN code with modules, for which, interfaces are made implicitly by the compiler. Appendix A illustrates how to use OpenACC functions to determine the total number of GPUs for run and how to select specific GPUs using their IDs. Choosing the device type (i.e. NVIDIA or AMD RADEON graphics) is also possible using the "acc_set_device_type()" which allows us to implement different device types on a cluster with heterogeneous installation of GPUs.

It is worth noting that besides MPI-OpenACC, there are other methods that implement multiple GPUs including asynchronous kernel computation using 1 CPU (i.e. master CPU–GPU) or multithreaded GPU using OpenMP (OpenMP + GPU), however, they prove to be less efficient alternatives [84].

Fig. 16 represents the multi-GPU speedup for the ACC-EVPCUFFT. We name the multi-GPU version as MPI-ACC-EVPCUFFT. With an increase in problem size from 16^3 to 128^3 and afterwards, the multi-GPU scalability is improved considerably. For the RVE resolution of 128^3, using 4 GPUs, speedups of 3.24x and 3.69x over the single GPU is achieved for the whole code and NR, respectively.

In order to simulate massive data sets (RVE sizes larger than 128^3), more GPUs are required to provide enough GPU dedicated memories for data allocation on the device. Running the code for RVE resolutions of 512^3 afterwards demands at least 220 GB of GPU memory which necessitates utilizing at least 20 GK210 chips of NVIDIA Tesla K80 (i.e. 10 T K80 GPUs) or 14 NVIDIA Tesla P100 GPUs to be able to run the code on GPU. Since these number of GPUs are not accessible on a single workstation, a cluster of GPU nodes should be utilized. In the next section we describe running our code on a distributed cluster of GPUs facilitating simulation of massive microstructure data sets.

10. MPI-ACC-EVPCUFFT benchmark on Cray Titan: Cluster of distributed GPU nodes

In order to benchmark the code on distributed GPU nodes, the Titan super computer [108] located at Oak Ridge National Laboratory (ORNL) is used to facilitate our crystal plasticity simulations. This supercomputer is equipped with NVIDIA Tesla K20X GPUs. Table 2 provides the hardware specs for NVIDIA Tesla K20X.

Note that NVIDIA Tesla K20X is older than its more recent peers K80 and P100 resulting in noticeable lower performance due to the hardware architecture. It is notable that a new supercomputer released by ORNL called Summit [109] uses Tesla Volta V100 GPUs instead of K20X. Performance benchmark of the code on this supercomputer is postponed for the future research when authors become users of Summit. It is important to mention that the MPI-FFTW3 is used for the FFT part to maintain the scalability while all other parts run on GPU using OpenACC. This is because running CUFFT concurrently on all nodes is not possible. A multi-GPU version of CUFFT called CUFFTXT is provided by NVIDIA CUDA toolkit [85], however, it is restricted to a maximum of 8 GPUs. This limitation hinders us to use it for desired number of GPU nodes. Running CUFFT on any desired number of devices using an alternative approach will be studied in a future research.

In order to study the RVE size effect on the elapsed CPU time, the EVPFFT solver for the RVE domains of 64^3, 128^3, and 256^3 is executed on 1, 8, and 64 NVIDIA Tesla K80 GPUs, respectively, while the FFTs run using MPI-FFTW. This version of implementation is termed MPI-ACC-EVPFFTW. Moving from RVE resolution

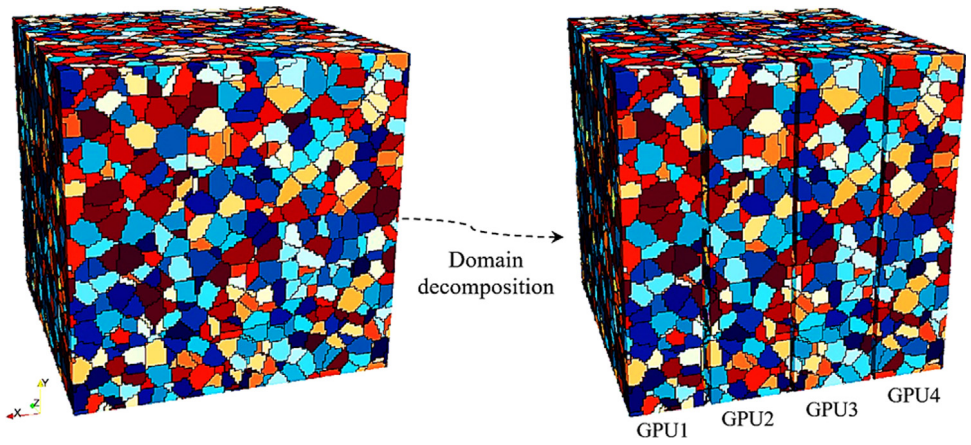

Fig. 15. RVE domain decomposition for the implementation of 4 GPUs using MPI. Every GPU works on its own chunk of data.

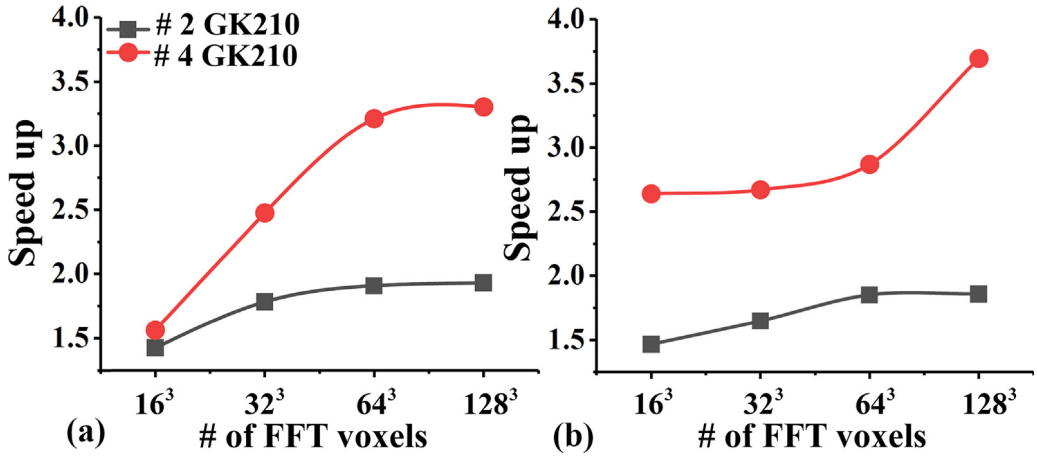

Fig. 16. MPI-ACC-EVPCUFFT speedup using 2 and 4 GK210 GPU chips of Tesla K80 (a) Total net scalability (b) NR scalability. With an increase in problem size from 16^3 to 128^3 and afterwards, the multi-GPU scalability is improved considerably. For the RVE resolution of 128^3, using 4 GPUs, speedups of 3.24x and 3.69x over the single GPU is achieved for EVPFFT (a) and NR (b), respectively.

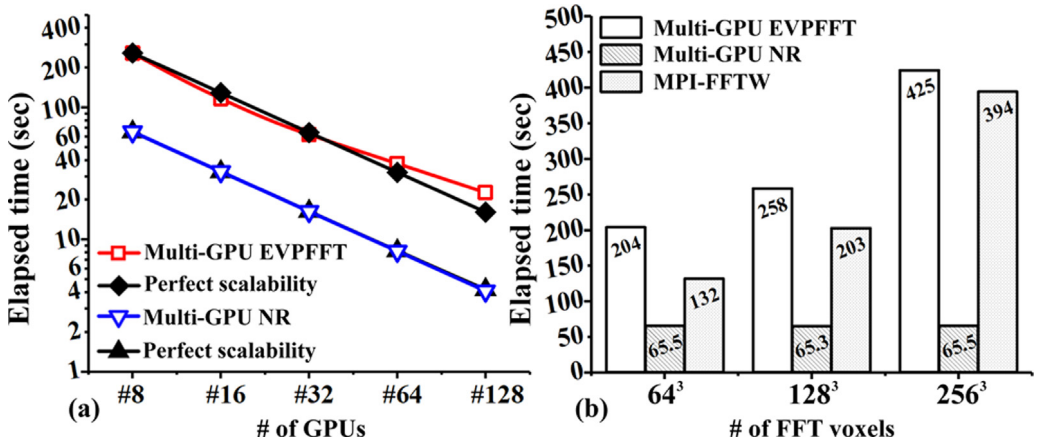

Fig. 17. Multi-GPU NR and multi-GPU EVPFFT scalabilities benchmarked on Titan Cray supercomputer utilizing distributed GPUs (a) Strong scalabilities of RVE resolution of 128^3 running on 8, 16, 32, 64, and 128 GPUs (b) Weak scalabilities of RVE resolutions of 64^3, 128^3, and 256^3 running on 1, 8, and 64 GPUs, respectively. Strong scalability of NR is perfect regardless of number of GPUs being utilized. The EVPFFT however starts to deviate slightly from perfect scalability using 64 and 128 GPUs and this is because the MPI-FFTW responsible for FFT calculations do not scale perfect on Titan after 64 processes for the RVE size of 128^3.

of 64^3 to 256^3, the total number of voxels is increased 8 times. Therefore, scaling the total number of GPUs to 8x elucidates how perfect the weak scalability of the Multi-GPU code is. In the case of perfect weak scalability, elapsed CPU time should be identical for all cases [110].

Fig. 17 represents multi-GPU NR and multi-GPU EVPFFT (i.e. MPI-ACC-EVPFFTW) strong and weak scalabilities benchmarked on Titan Cray supercomputer utilizing distributed GPUs running simple compression of Cu for 10 strain increments under applied strain rate of 0.001 s^{-1}. RVE resolution of 128^3 running

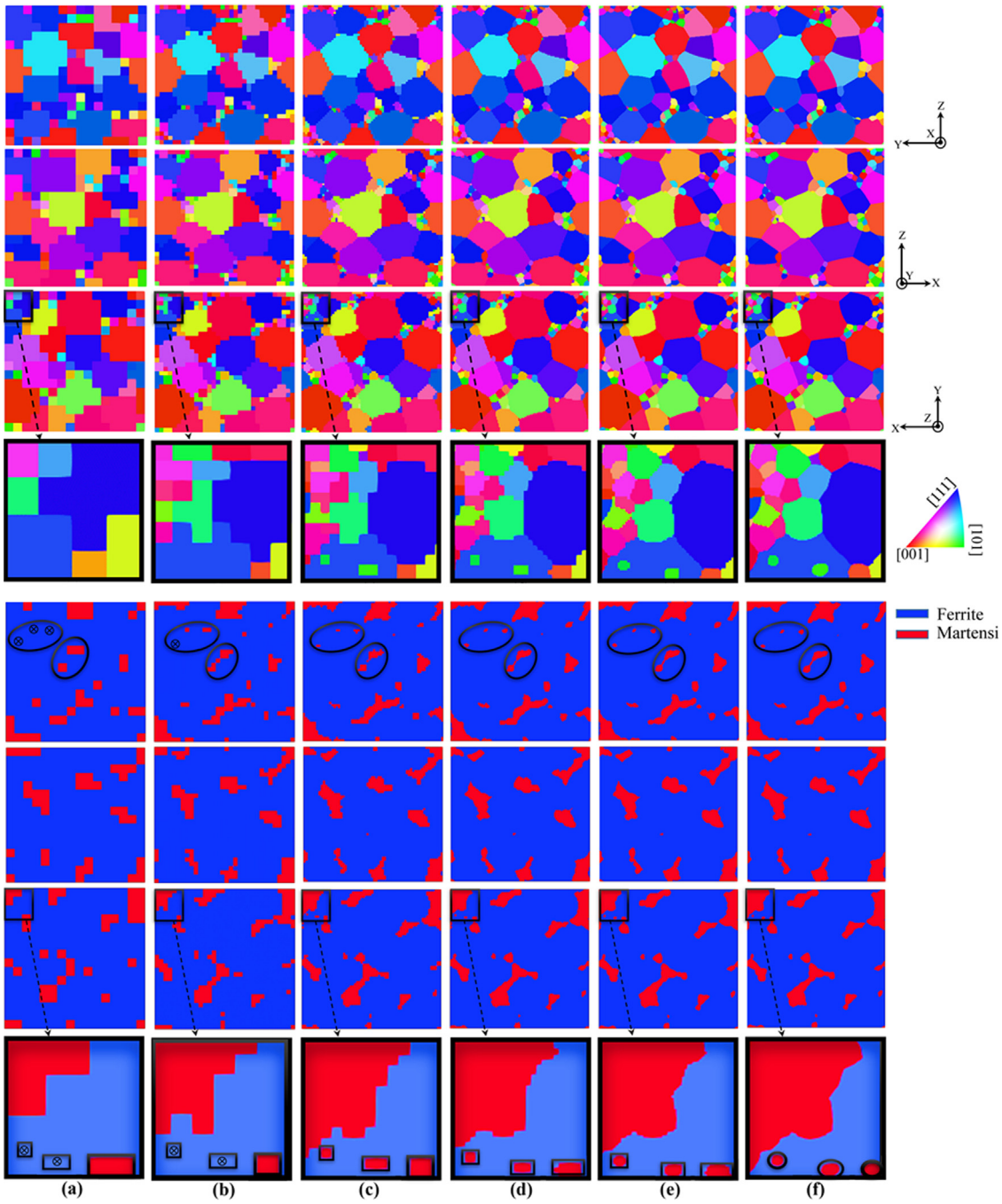

Fig. 18. Effect of RVE voxel resolution in capturing small martensitic islands within ferrite in DP 590: (a) 16^3 (b) 32^3 (c) 64^3 (d) 128^3 (e) 256^3 (f) 512^3. The first 4 rows show the grain structure, while the second 4 rows indicate the phase fields for ferrite and martensite. For resolutions lower than 256^3, the grain/phase boundaries become jagged which is physically spurious. Moreover, in RVEs smaller than 64^3, the small features disappear hindering us from representing the microstructure accurately. The DREAM3D software package is used to generate the synthetic microstructure.

on 8, 16, 32, 64, and 128 GPUs were implemented for strong scalability benchmark. Strong scalability of NR is perfect regardless of number of GPUs being utilized. The EVPFFT however starts to deviate slightly from perfect scalability using 64 and 128 GPUs and this is because the MPI-FFTW responsible for FFT calculations does not scale perfect on Titan after 64 MPI processes for the RVE size of 128^3. This analogy holds true for the weak scalability as well. The NR weak scalability is perfect (i.e. almost identical elapsed CPU times for all configurations), however, since the FFTs scale as order of $n\ \log(n)$ (i.e. where n is the RVE size), moving to larger RVE resolutions does not scale linear resulting in growing trend for the multi-GPU EVPFFT running on MPI-FFTW.

11. Application of ACC-EVPCUFFT to resolving fields in a dual phase steel DP 590 microstructure

To demonstrate another utility of the implementation we use it for resolving fine microstructural features. A large RVE sizes are desirable for understanding behavior of multi-phase alloys. Understanding the strain and stress gradients varying at interfaces is crucial for performance design of polycrystalline multi-phase alloys. In a ferrite-martensitic dual phase steel, the master phase is ferrite matrix in which martensitic phase in a reinforcement. Phase fractions depend on the alloy type. We study the DP 590 in which the martensitic phase fraction is 7.7% [111,112]. In addition

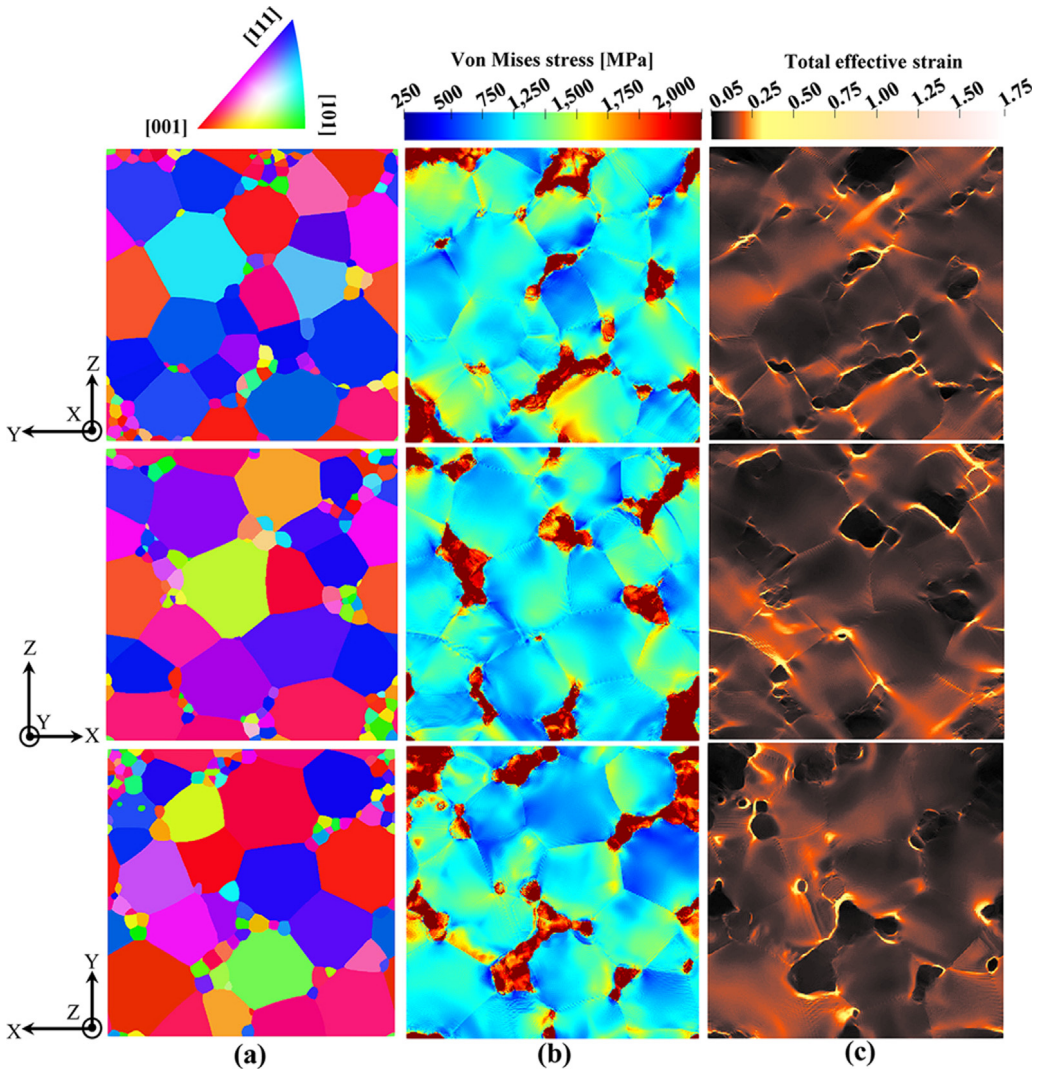

Fig. 19. Stress and strain fields over DP 590 microstructure predicted by the ACC-EVPCUFFT model for an RVE size of 256^3.

to phase fraction, an attempt is made to create a microstructure resembling the ratio in grain size. We show the effect of RVE resolution and its significant importance in capturing the very small martensitic grains more realistically. Moreover, we show how resolution is vital for defining the grain boundaries to be smooth. Fig. 18 presents the effect of RVE voxel resolution going from 16^3 to 512^3. The DREAM3D software package [113] is used to generate the synthetic microstructure. It is clear that for resolutions lower than 256^3, the grain/phase boundaries become jagged and not realistic. Moreover, in RVEs smaller than 64^3, the small features of the martensitic phase disappear hindering us from representing the dual phase microstructure accurately.

Fig. 19 represents the stress and strain gradients at inter-granular boundaries of ferrite and martensite phases in DP 590 steel under monotonic simple compression up to a macroscopic strain of 0.16 with applied strain rate of 0.001 s^{-1}, captured by ACC-EVPCUFFT using RVE size of 256^3. Such resolution is mandatory to capture the very sharp and narrow field gradients observed around the Martensitic islands. More careful examination of the figure reveals very detailed traces of stress and strain concentrations (hotspots) indicating the crack incubation regions and sources of inter-granular fracture. These features cannot be captured unless a very high resolution is used. Note that we are not able to run the RVE resolution of 512^3 due to insufficient memory per node while reading/initializing the data concurrently on all MPI processes. A Parallel I/O implementation would help which is going to be addresses in future research.

12. A flowchart summarizing the developed parallel implementations of the EVPFFT solver

In order to sum up all of the parallel implementations of the EVPFFT solver developed so far, a comprehensive flowchart is provided in Fig. 20, illustrating the flow of the code for all parallel implementations including the features from previous work [56] as well. This schematic will help the reader significantly to review all the performance improvements presented here in a nut shell.

13. Conclusions

In this work, we develop a computationally efficient implementation of a full-field crystal plasticity solver for predicting micromechanical behavior of crystalline materials taking advantages of GPUs. While porting the NR subroutine on GPU, it was found that using GJE in NR solver results in an improvement over the LU decomposition because the GJE linear equation solver suppresses sequential runs on GPU threads. Next, the GPU implementation executes the FFT calculations using the CUFFT library

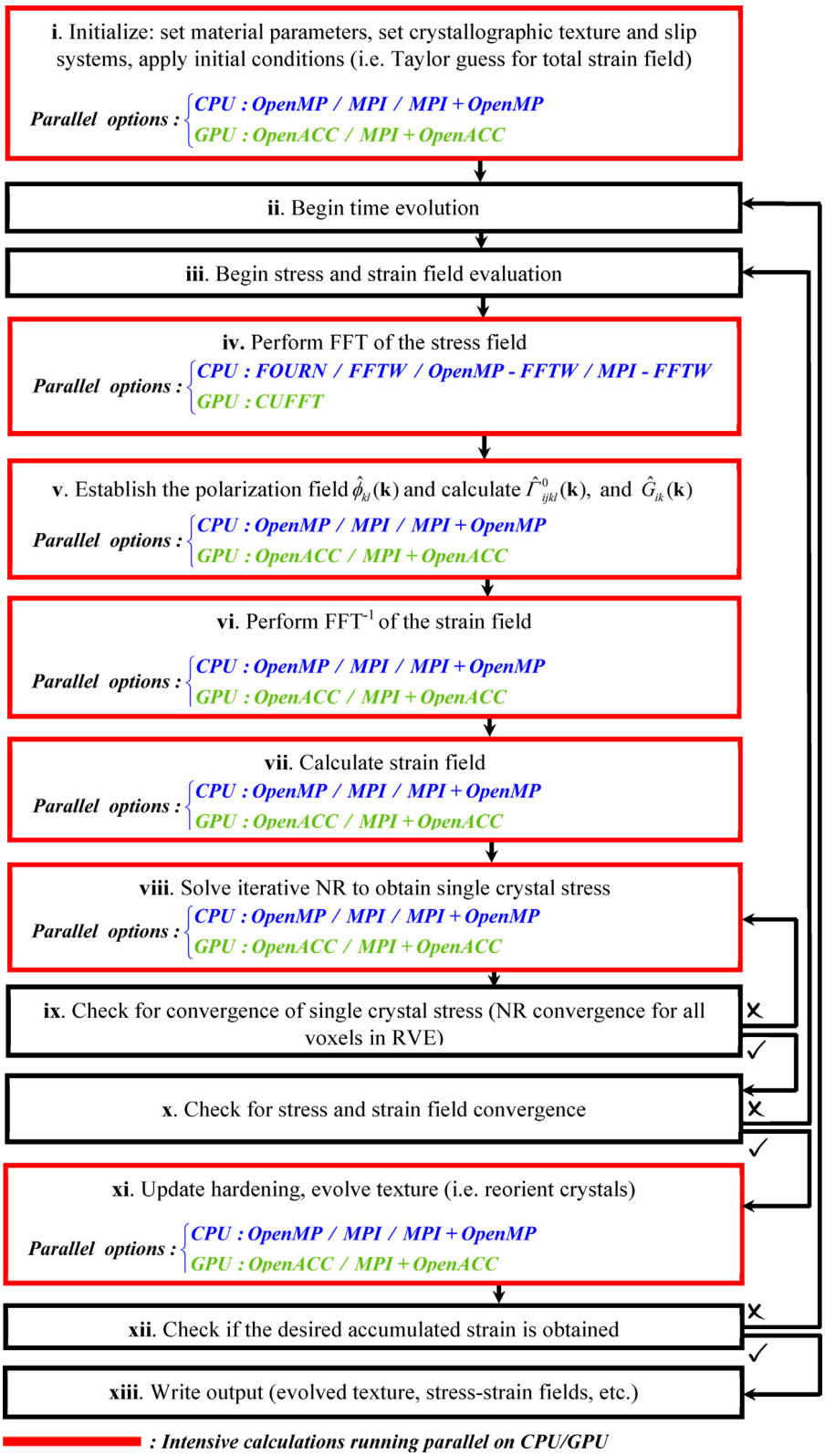

Fig. 20. All-in-one flowchart summarizing the developed parallel implementations of the EVPFFT model.

utilizing CUDA resulting in further computational gains. Furthermore, the remaining code is accelerated on GPU, except the I/O subroutines to ensure minimal data transfer between CPU and GPU. Finally, a combination of MPI through the domain decomposition for multi-GPU execution and the OpenACC standard is exploited for the multi-GPU version of the implementation. The overall implementations are termed ACC-EVPCUFFT for single GPU and MPI-ACC-EVPCUFFT for multiple GPUs. Significant speedups over the original EVPFFT model and the recent MPI-EVPFFTW are achieved using the GPU versions on multi core computer workstations and Cray Titan cluster. The GPU performance benchmarks were carried out on NVIDIA Tesla Pascal and

Kepler architectures (i.e. Tesla P100 and Tesla K80). For example, using a single P100 GPU on a single CPU core outperforms running the code with MPI decomposition over 64 Intel Xeon 2695 v4 processes for the RVE size of 128^3.

Having the multi-GPU version of the code provides us with the capability to run high performance crystal plasticity simulations involving high resolution microstructures i.e. tens of millions of FFT sampling points for resolving local mechanical fields and microstructural evolution highly efficiently on an appropriate hardware. Additionally, the performance gains represent a significant incentive to integrating these models into macroscopic FE computational tools to enable more accurate simulations of material behavior during metal forming processes and in-service conditions. The implementation will be a CPU–GPU hybrid, where finite element mesh is domain distributed over CPUs and underneath microstructural cell calculations are performed on GPUs.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

This research was sponsored by the U.S. National Science Foundation and was accomplished under the CAREER, USDA Grant No. CMMI-1650641.

Appendix A. Openacc functions for multi-GPU framework setup

The functions "acc_get_num_devices" and "acc_get_device _num" are used to determine the total number of available GPUs and the GPU number (i.e. device id), respectively. The function "acc_set_device_num" sets the device id (i.e. GPU number) for the current MPI process (i.e. selects the GPU to be controlled by the specific CPU defined by MPI_rank). For example, if we consider 4 GK210 GPU chips of 2 T K80 graphics cards, running the program built by the PGI MPI compiler (i.e. mpif90) using the "mpirun –n 4 ./executable" command initializes 4 CPU cores each responsible to control one of the 4 GPUs. Note that since the GPU id is set by MPI rank, then the device ids of 0, 1, 2, and 3 account for GPU numbers 1, 2, 3, and 4, respectively.

```
USE mpi
USE openacc
call MPI_COMM_SIZE(MPI_COMM_WORLD, size, ierror)
call MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierror)
num_devices=acc_get_num_devices(acc_device_nvidia)
devicenum=mod(rank,num_devices)
call  acc_set_device_num(devicenum,acc_device_nvidia)
 if(rank.eq.0) then
  print*, "Number Of GPUs:",num_devices
endif
devicenum=acc_get_device_num(acc_device_nvidia)
print*,"#CPU rank:",rank,"GPU number:",devicenum
print*, "Number Of GPUs:",num_devices
call acc_set_device_num(0,acc_device_nvidia)
call acc_init(acc_device_nvidia)
```

Appendix B. Supplementary data

Supplementary material related to this article can be found online at https://doi.org/10.1016/j.cpc.2020.107231.

References

[1] W.F. Hosford, R.M. Caddell, Metal Forming: Mechanics and Metallurgy, Cambridge University Press, New York, USA, 2011.
[2] M. Jahedi, M. Knezevic, M. Paydar, J. Mater. Eng. Perform. 24 (2015) 1471–1482.
[3] M. Jahedi, I.J. Beyerlein, M.H. Paydar, M. Knezevic, Adv. Energy Mater. 20 (2018) 1600829.
[4] T.J. Barrett, M. Knezevic, Int. J. Mech. Sci. 174 (2020) 105508.
[5] C.M. Poulin, Y.P. Korkolis, B.L. Kinsey, M. Knezevic, Mater. Des. 161 (2019) 95–105.
[6] M. Knezevic, C.M. Poulin, X. Zheng, S. Zheng, I.J. Beyerlein, Mater. Sci. Eng. A 758 (2019) 47–55.
[7] N. Barton, J. Bernier, J. Knap, A. Sunwoo, E. Cerreta, T. Turner, Internat. J. Numer. Methods Engrg. 86 (2011) 744–764.
[8] J.H. Panchal, S.R. Kalidindi, D.L. McDowell, Comput. Aided Des. 45 (2013) 4–25.
[9] T.J. Barrett, D.J. Savage, M. Ardeljan, M. Knezevic, Comput. Mater. Sci. 141 (2018) 269–281.
[10] M. Jahedi, M.H. Paydar, M. Knezevic, Mater. Charact. 104 (2015) 92–100.
[11] T.J. Barrett, A. Eghtesad, R.J. McCabe, B. Clausen, D.W. Brown, S.C. Vogel, M. Knezevic, Materialia 6 (2019) 100328.
[12] O. Diard, S. Leclercq, G. Rousselier, G. Cailletaud, Comput. Mater. Sci. 25 (2002) 73–84.
[13] M. Jahedi, M. Ardeljan, I.J. Beyerlein, M.H. Paydar, M. Knezevic, J. Appl. Phys. 117 (2015) 214309.
[14] M. Ardeljan, R.J. McCabe, I.J. Beyerlein, M. Knezevic, Comput. Methods Appl. Mech. Engrg. 295 (2015) 396–413.
[15] Z. Zhao, M. Ramesh, D. Raabe, A.M. Cuitiño, R. Radovitzky, Int. J. Plast. 24 (2008) 2278–2297.
[16] M. Ardeljan, I.J. Beyerlein, M. Knezevic, Int. J. Plast. 99 (2017) 81–101.
[17] M. Ardeljan, M. Knezevic, Acta Mater. 157 (2018) 339–354.
[18] M. Ardeljan, M. Knezevic, T. Nizolek, I.J. Beyerlein, N.A. Mara, T.M. Pollock, Int. J. Plast. 74 (2015) 35–57.
[19] M. Knezevic, M.R. Daymond, I.J. Beyerlein, Scr. Mater. 121 (2016) 84–88.
[20] F. Feyel, Comput. Methods Appl. Mech. Engrg. 192 (2003) 3233–3244.
[21] B. Liu, D. Raabe, F. Roters, P. Eisenlohr, R.A. Lebensohn, Modelling Simulation Mater. Sci. Eng. 18 (2010) 085005.
[22] A. Prakash, R.A. Lebensohn, Modelling Simulation Mater. Sci. Eng. 17 (2009) 064010.
[23] T. Mura, Micromechanics of Defects in Solids, second ed., Martinus Nijhoff Publishers, Netherlands, 1987.
[24] R. Lebensohn, Acta Mater. 49 (2001) 2723–2737.
[25] R.A. Lebensohn, A.K. Kanjarla, P. Eisenlohr, Int. J. Plast. 32–33 (2012) 59–69.
[26] B. Mihaila, M. Knezevic, A. Cardenas, Internat. J. Numer. Methods Engrg. 97 (2014) 785–798.
[27] D.J. Savage, M. Knezevic, Comput. Mech. 56 (2015) 677–690.
[28] Y. Mellbin, H. Hallberg, M. Ristinmaa, Internat. J. Numer. Methods Engrg. 100 (2014) 111–135.
[29] K. Chockalingam, M.R. Tonks, J.D. Hales, D.R. Gaston, P.C. Millett, L. Zhang, Comput. Mech. 51 (2013) 617–627.
[30] M. Knezevic, S.R. Kalidindi, Comput. Mater. Sci. 39 (2007) 643–648.
[31] M. Knezevic, S.R. Kalidindi, D. Fullwood, Int. J. Plast. 24 (2008) 1264–1276.
[32] M. Knezevic, S.R. Kalidindi, R.K. Mishra, Int. J. Plast. 24 (2008) 327–342.
[33] H.K. Duvvuru, M. Knezevic, R.K. Mishra, S.R. Kalidindi, Mater. Sci. Forum 546 (2007) 675–680.
[34] N. Landry, M. Knezevic, Materials 8 (2015) 6326–6345.
[35] B.S. Fromm, B.L. Adams, S. Ahmadi, M. Knezevic, Acta Mater. 57 (2009) 2339–2348.
[36] J.B. Shaffer, M. Knezevic, S.R. Kalidindi, Int. J. Plast. 26 (2010) 1183–1194.
[37] S.R. Kalidindi, H.K. Duvvuru, M. Knezevic, Acta Mater. 54 (2006) 1795–1804.
[38] M. Knezevic, N.W. Landry, Mech. Mater. 88 (2015) 73–86.
[39] M. Knezevic, H.F. Al-Harbi, S.R. Kalidindi, Acta Mater. 57 (2009) 1777–1784.
[40] H.F. Al-Harbi, M. Knezevic, S.R. Kalidindi, CMC: Comput. Mater. Contin. 15 (2010) 153–172.
[41] M. Zecevic, R.J. McCabe, M. Knezevic, Mech. Mater. 84 (2015) 114–126.
[42] M. Zecevic, R.J. McCabe, M. Knezevic, Int. J. Plast. 70 (2015) 151–165.
[43] M. Knezevic, D.J. Savage, Comput. Mater. Sci. 83 (2014) 101–106.
[44] M. Jahedi, E. Ardjmand, M. Knezevic, Powder Technol. 311 (2017) 226–238.
[45] N.R. Barton, J. Knap, A. Arsenlis, R. Becker, R.D. Hornung, D.R. Jefferson, Int. J. Plast. 24 (2008) 242–266.
[46] N.R. Barton, J.V. Bernier, R.A. Lebensohn, D.E. Boyce, Comput. Methods Appl. Mech. Engrg. 283 (2015) 224–242.
[47] M. Ardeljan, I.J. Beyerlein, B.A. McWilliams, M. Knezevic, Int. J. Plast. 83 (2016) 90–109.

[48] W.G. Feather, S. Ghorbanpour, D.J. Savage, M. Ardeljan, M. Jahedi, B.A. McWilliams, N. Gupta, C. Xiang, S.C. Vogel, M. Knezevic, Mechanical. response, twinning, Int. J. Plast. 120 (2019) 180–204.
[49] M. Knezevic, R.J. McCabe, R.A. Lebensohn, C.N. Tomé, C. Liu, M.L. Lovato, B. Mihaila, J. Mech. Phys. Solids 61 (2013) 2034–2046.
[50] M. Knezevic, R.A. Lebensohn, O. Cazacu, B. Revil-Baudard, G. Proust, S.C. Vogel, M.E. Nixon, Mater. Sci. Eng. A 564 (2013) 116–126.
[51] M. Knezevic, A. Levinson, R. Harris, R.K. Mishra, R.D. Doherty, S.R. Kalidindi, Acta Mater. 58 (2010) 6230–6242.
[52] M. Zecevic, I.J. Beyerlein, M. Knezevic, Int. J. Plast. 93 (2017) 187–211.
[53] M. Zecevic, M. Knezevic, JOM 69 (2017) 922–929.
[54] M. Zecevic, M. Knezevic, Mech. Mater. 136 (2019) 103065.
[55] T.J. Barrett, M. Knezevic, Comput. Methods Appl. Mech. Engrg. 354 (2019) 245–270.
[56] A. Eghtesad, T.J. Barrett, K. Germaschewski, R.A. Lebensohn, R.J. McCabe, M. Knezevic, Adv. Eng. Softw. 126 (2018) 46–60.
[57] M. Frigo, S.G. Johnson, FFTW: Fastest Fourier transform in the west, Astrophysics Source Code Library, 2012.
[58] J. Nickolls, W.J. Dally, IEEE Micro 30 (2010).
[59] A. Eghtesad, M. Knezevic, Comput. Part. Mech. 5 (2018) 387–409.
[60] A. Eghtesad, K. Germaschewski, I.J. Beyerlein, A. Hunter, M. Knezevic, Adv. Eng. Softw. 115 (2018) 248–267.
[61] J.D. Owens, M. Houston, D. Luebke, S. Green, J.E. Stone, J.C. Phillips, Proc. IEEE 96 (2008) 879–899.
[62] V. Tvergaard, Int. J. Fract. 18 (1982) 237–252.
[63] P. Bauer, V. Klement, T. Oberhuber, V. Žabka, Comput. Phys. Comm. 200 (2016) 50–56.
[64] 2011-2014 OpenACC.org, http://www.openacc-standard.org/.
[65] C. Nvidia, CUFFT Library, Version, 2010.
[66] R.A. Lebensohn, A.K. Kanjarla, P. Eisenlohr, Int. J. Plast. 32 (2012) 59–69.
[67] M. Knezevic, M. Zecevic, I.J. Beyerlein, R.A. Lebensohn, Comput. Methods Appl. Mech. Engrg. 308 (2016) 468–482.
[68] R. Bellman, G. Adomian, Green's Functions for Partial Differential Equations, Partial Differential Equations, Springer, 1985, pp. 243–247.
[69] A.I. Zayed, IEEE Signal Process. Lett. 5 (1998) 101–103.
[70] J. Michel, H. Moulinec, P. Suquet, Internat. J. Numer. Methods Engrg. 52 (2001) 139–160.
[71] A. Eghtesad, M. Zecevic, R.A. Lebensohn, R.J. McCabe, M. Knezevic, Comput. Mech. 61 (2018) 89–104.
[72] M. Knezevic, B. Drach, M. Ardeljan, I.J. Beyerlein, Comput. Methods Appl. Mech. Engrg. 277 (2014) 239–259.
[73] A. Eghtesad, T.J. Barrett, M. Knezevic, Acta Mater. 155 (2018) 418–432.
[74] M.A. Meyers, K.K. Chawla, Mechanical Behavior of Materials, Prentice Hall, Upper Saddle River, New Jersey, 1998.
[75] C. Tomé, R. Lebensohn, Manual for Code Visco-Plastic Self-Consistent (Vpsc), Los Alamos National Laboratory, New Mexico, USA, 2009.
[76] A. Eghtesad, M. Knezevic, J. Mech. Phys. Solids 134 (2020) 103750.
[77] S. Ghorbanpour, M. Zecevic, A. Kumar, M. Jahedi, J. Bicknell, L. Jorgensen, I.J. Beyerlein, M. Knezevic, Int. J. Plast. 99 (2017) 162–185.
[78] S. Ghorbanpour, M.E. Alam, N.C. Ferreri, A. Kumar, B.A. McWilliams, S.C. Vogel, J. Bicknell, I.J. Beyerlein, M. Knezevic, Int. J. Plast. 125 (2020) 63–79.
[79] User's Guide for x86-64 CPUs - PGI Compilers, Version 2016.
[80] CAPS openACC compiler the fastest way to many-core programming, 2012, http://www.caps-entreprise.com.
[81] Openacc accelerator directives, 2012, http://www.training.praceri.eu/uploads/txpracetmo/OpenACC.pdf.
[82] B. Lebacki, M. Wolfe, D. Miles, The Pgi Fortran and C99 Openacc Compilers, Cray User Group, 2012.
[83] J. Herdman, W. Gaudin, S. McIntosh-Smith, M. Boulton, D.A. Beckingsale, A. Mallinson, S.A. Jarvis, Accelerating Hydrocodes with OpenACC, OpenCL and CUDA, High Performance Computing, Networking, Storage and Analysis (SCC), 2012 SC Companion, IEEE, 2012, pp. 465–471.
[84] R. Farber, Parallel Programming with OpenACC, Newnes, 2016.
[85] 2007-2014 NVIDIA Corporation, CUDA Toolkit Documentation v6.5, http://docs.nvidia.com/cuda/#axzz3MMC3iZGv.
[86] C. Nvidia, Cublas Library, Vol. 15, NVIDIA Corporation, Santa Clara California, 2008, p. 27.
[87] The openacc standard, 2016, http://www.openacc-standard.org.
[88] W. Press, B. Flannery, S. Teukolsky, W. Vetterling, Gauss-Jordan Elimination and Gaussian Elimination with Backsubstitution, Sections in Numerical Recipes in Fortran: The Art of Scientific Computing, 1992.
[89] G. Peters, J.H. Wilkinson, Commun. ACM 18 (1975) 20–24.
[90] R. Melhem, Parallel Comput. 4 (1987) 339–343.
[91] P.S. Stanimirović, M.D. Petković, Appl. Math. Comput. 219 (2013) 4667–4679.
[92] A.N. Malyshev, Computing 65 (2000) 281–284.
[93] I. Buck, Nvidia'S Next-Gen Pascal GPU Architecture to Provide 10x Speedup for Deep Learning Apps, 2015.
[94] W.H. Press, S.A. Teukolsky, W.T. Vetterling, B.P. Flannery, Numerical Recipes in C, Cambridge Univ Press, 1982.
[95] H. William, Numerical Recipes in Fortran, Cambridge University Press, 1992.
[96] M. Frigo, S.G. Johnson, Proceedings of the 1998 IEEE International Conference on Acoustics, Speech and Signal Processing, 1998, IEEE, 1998, pp. 1381–1384.
[97] M. Frigo, S.G. Johnson, FFTW User'S Manual, Massachusetts Institute of Technology, 1999.
[98] M. Frigo, S.G. Johnson, Fastest F Ourier T Ransform in the W Est, 2006.
[99] M. Frigo, S.G. Johnson, The Fastest Fourier Transform in the West, DTIC Document, 1997.
[100] C. Nvidia, Compute Unified Device Architecture Programming Guide, 2007.
[101] NVIDIA Corporation. NVIDIA CUDA Compute Unified Device Architecture Programming Guide. http://developer.nvidia.com/object/cuda.html.
[102] R. Dolbeau, Theoretical Peak FLOPS Per Instruction Set on Less Conventional Hardware, 2015.
[103] C. Nvidia, Toolkit Documentation NVIDIA CUDA Getting Started Guide for Linux, 2014.
[104] NVIDIA Tesla V100, 2017.
[105] A. Pajankar, Message Passing Interface, Raspberry Pi Supercomputing and Scientific Programming, Springer, 2017, pp. 61–65.
[106] D.W. Walker, J.J. Dongarra, Supercomputer 12 (1996) 56–68.
[107] F.C.T. Herault, J. Dongarra, Recent Advances in Parallel Virtual Machine and Message Passing Interface.
[108] A.L. Shimpi, Inside the titan supercomputer: 299k amd x86 cores and 18.6 k nvidia gpus, AnandTech online computer hardware magazine, October, (2012).
[109] J. Hines, Comput. Sci. Eng. 20 (2018) 78–82.
[110] M.T. Heath, Int. J. High Perform. Comput. Appl. 29 (2015) 320–330.
[111] M. Zecevic, Y.P. Korkolis, T. Kuwabara, M. Knezevic, J. Mech. Phys. Solids 96 (2016) 65–87.
[112] A.M. Cantara, M. Zecevic, A. Eghtesad, C.M. Poulin, M. Knezevic, Int. J. Mech. Sci. 151 (2019) 639–649.
[113] M.A. Groeber, M.A. Jackson, Integr. Mater. Manuf. Innov. 3 (2014) 5.