Simulation of Supernova Explosion Accelerated on GPU: Spherically Symmetric Neutrino-Radiation Hydrodynamics

Hideo Matsufuru[1](✉) and Kohsuke Sumiyoshi[2]

[1] High Energy Accelerator Research Organization (KEK), 1-1 Oho, Tsukuba, Ibaraki 305-0801, Japan
hideo.matsufuru@kek.jp

[2] National Institute of Technology, Numazu College, 3600 Ooka, Numazu, Shizuoka 410-8501, Japan
sumi@numazu-ct.ac.jp

Abstract. Performing large scale numerical simulations is essential to understand the explosion mechanism of core-collapse supernovae. It is mandatory to solve a multi-physics system described by coupled equations of hydrodynamics and neutrino-radiation transfer in multi-dimensions. Since the neutrino transfer is in principle governed by the Boltzmann equation in six-dimensional space, numerical simulations require large computational resources. Having the final goal to realize acceleration of such simulations, we study the numerical computations of neutrino-radiation hydrodynamics under spherical symmetry by exploiting GPU devices, which has become a powerful equipment in scientific high performance computing. As the first step, we focus on the most time consuming part: a linear equation solver for a block tridiagonal matrix which appears in a spherically symmetric simulation. To offload this part to the GPU devices, we employ OpenACC as a framework of implementation as well as make use of cuBLAS library that is available on NVIDIA's CUDA environment. Practical performance is examined on two systems with the Kepler and Pascal generations of NVIDIA's GPU architecture.

1 Introduction

Supernova explosions are spectacular displays in the Universe and the dynamical phenomena at the end of stellar life [1,2]. Gravitational collapse of the massive stars with more than 10 times the solar mass leads to the compression of Fe core and the following core bounce, which launches the shock wave toward the explosion. Despite the long history of investigations based on this general idea, the detailed mechanism of the core-collapse supernova explosion is still elusive, because only prohibitively large scale numerical simulations can explore the

O. Gervasi et al. (Eds.): ICCSA 2018, LNCS 10962, pp. 440–455, 2018.
https://doi.org/10.1007/978-3-319-95168-3_30

multi-dimensional mechanism with multi-physics in multi-scale. Precise understanding of the supernova explosion is one of the essential issues in astrophysics in order to reveal the origin of heavy elements, the formation of compact objects (neutron stars or black holes), and observation of neutrino bursts and gravitational waves.

Study of the explosion mechanism of core-collapse supernovae is challenging since all the four fundamental interactions play important roles as physical processes. The strong and electromagnetic interactions govern the properties of dense matter in the stellar dynamics under the gravitational influence in general relativity. The weak interaction is essential to describe the production, extinction and scattering of neutrinos as one of essential composition in dense matter. The neutrinos are agents of reservoir and transport of energy in the supernova core and play a crucial role in the explosion mechanism. During gravitational collapse, the stellar core becomes opaque to neutrinos due to high densities. Neutrinos are once trapped in the central dense object and later gradually emitted and contribute to the revival of stalled shock wave by heating due to the absorption of material. This transport of neutrinos in stellar dynamics is non-trivial and is governed by coupled equations of radiation transfer and hydrodynamics. Neutrino-radiation hydrodynamics is commonly known as a difficult problem similar to the radiation hydrodynamics in physics and engineering with high performance computing.

So far most of large scale simulations have been performed on massively parallel clusters, such as the K computer. The Intel Xeon Phi is also included in this kind of architecture. Recently another type of architecture, which makes use of arithmetic accelerators such as GPU, has become popular in high performance computing. Although the accelerator device has an advantage in cost performance, the code implementation becomes much more involved to achieve desired performance. Furthermore, whether accelerators work efficiently depends strongly on the structure of numerical algorithms.

In this work, we apply the accelerator architecture to the supernova simulations. As the first step in this direction, we consider a spherically symmetric system, namely spatially one-dimensional simulations. This is because precise numerical simulations under the spherical symmetry, albeit it shows no explosion, are important as the basis of multi-dimensional simulations. First principle calculations under the spherical symmetry are strongly demanded to provide the reliable data of neutrino emission for observations as well as for formation of compact objects. The spherical modeling is also used as initial models for multi-dimensional simulations and for proto-neutron star cooling. Systematic survey of gravitational collapse of massive stars with various modeling of stellar evolution, examination of nuclear physics and neutrino physics are often done by the spherical simulations. Therefore acceleration of the spherical simulations is in general beneficial to produce many models for supernova studies.

The dynamics of neutrinos and dense matter are described by the Boltzmann equation and hydrodynamic equations, respectively, that are coupled to each other. The implicit scheme is adopted for stiff equations of time evolution with largely different time scales. In the current study, we focus on the linear equation to update the variables which appears in every step of the time evolution in the implicit scheme. This part is chosen as the first target of offloading to GPU since a large portion of computing time is consumed by an iterative solver for this linear equation. We apply OpenACC after changing the data layout and loop structure in an alternative code to the original one. We measure the performance of the code on two GPU systems at several system sizes. Our study is a step toward the goal to realize multi-dimensional simulations by extending our approach in one-dimension since the computational loads have similar patterns.

This paper is organized as follows. Next section summarizes the formulation to investigate the supernova explosion. After summarizing the numerical simulation scheme of supernovae adopted in this work, we describe the linear equation that is to be solved on accelerator devices. In Sect. 3 we describe our implementation of the code for GPU computation and show the result of performance in Sect. 4. Section 5 is devoted to our conclusion and future prospects.

2 Formulation

2.1 Overview of Supernova Simulations

The numerical study of core-collapse supernovae has long history over half-century [1]. The system of equations for hydrodynamics with gravitational force and neutrino transfer must be solved to follow the time evolution of dynamics of stellar matter, compositional change and neutrino distributions. Data of physics processes such as sets of equation of state and reaction rates for neutrinos are implemented in the numerical simulations. It is mandatory to treat the system under the general relativistic description. Among the difficulties of solving the system, the neutrino transfer is the most problematic and remains challenging in terms of high performance computing. Since the basic equation of neutrino transport is the Boltzmann equation, one has to solve the time evolution of neutrino distribution function in six dimensions (three dimensions in space and three dimensions in neutrino momentum space) [3]. It is to be noted that angle and energy distributions of neutrinos are essential since the neutrino distributions evolve under non-equilibrium conditions in largely different time scales from hydrodynamics with angle and energy-dependent reaction rates. Full treatment of the 6D neutrino transport and 3D Hydrodynamics is in principle ideal to reveal the explosion dynamics in 3D. However, necessary computing resources are prohibitive and its realization demands large-scale numerical costs that have been not practical even on recent supercomputers. A study of neutrino transfer in 3D has been made for stationary situation of supernova cores [3].

For this reason, numerical studies of supernovae have remarkable progress along with the rapid growth of supercomputing power. In its infancy of numerical studies, simple approximations and/or assumptions were used to model the system. Approximate methods such as diffusion approximations have been improved or removed in the progress of later researches. Only recently in the last decade, the first principle calculation under the spherical symmetry is achieved [4,5].

The assumption of spherical symmetry in space reduces the number of degrees of freedom of the Boltzmann equation to three (radial coordinate and neutrino energy and angle). Numerical simulations of general relativistic neutrino-radiation hydrodynamics became possible and have been performed to explore the spherical dynamics of core-collapse supernovae. Such studies have been utilized to demonstrate that no explosion generally occurs under the spherical geometry, under which the shock wave stalls after the launch due to the energy loss except for particular circumstances.

Multi-dimensional dynamics in combination with neutrino heating, therefore, is believed to be essential to achieve successful supernova explosions. In numerical simulations in two and three dimensions, certain approximations are made to perform many models for systematic studies. Although numerical simulations by directly solving the Boltzmann equation have been achieved recently, it costs computational resources of multi-million node-hour on the K computer for the limited set of models still in two dimensions [6]. Therefore, it is indispensable to drastically accelerate the computation to perform systematic modeling and to extend them to modeling in three dimensions.

In the following subsections, we briefly summarize the formulations and corresponding computations in this work. We assume the geometry under spherical symmetry in the current study and describe some equations in a simplified form to explain the ingredients.

2.2 Basic Equations

Hydrodynamics. The basic equations consist of the Einstein equations

$$G^{\mu\nu} = 8\pi T^{\mu\nu}, \tag{1}$$

and the Euler equation for the comoving fluid element

$$\nabla_\nu T^{\mu\nu} = 0, \tag{2}$$

with the baryon number conservation equation

$$\nabla_\nu (\rho_b u^\nu) = 0, \tag{3}$$

under a certain metric of spacetime $g_{\mu\nu}$ in general relativity [7]. The energy-momentum tensor for an ideal fluid is

$$T^{\mu\nu} = [\rho_b(1+\epsilon) + p]u^\mu u^\nu - p g_{\mu\nu} + T^{\mu\nu}_\nu, \tag{4}$$

where ρ_b is baryon mass density, u^μ is the four-velocity of the matter, ϵ and p are respectively the specific internal energy and pressure, and $T^{\mu\nu}_\nu$ is the neutrino component of energy-momentum tensor. $G^{\mu\nu}$ is the Einstein tensor.

These equations correspond to the formulation for the time evolution of the fluid dynamics with exchange of energy-momentum due to neutrinos, the baryon conservation and the compositional change. There are additional basic equations for metric components for the general relativistic description. The equation of thermodynamics is solved simultaneously to follow the time evolution of entropy for handling the equation of state. Thus dynamical variables of fluid elements for general relativistic hydrodynamics in the Lagrangian scheme are the following: radial position, radial velocity, density, internal energy, specific enthalpy, entropy, electron fraction, gravitational mass, and three general relativistic variables.

Boltzmann Equation. The evolution of neutrino distribution is described by the general relativistic Boltzmann equation

$$\frac{dx^{\mu}}{d\tau}\frac{\partial f_{\nu}}{\partial x^{\mu}} + \frac{dp^{i}}{d\tau}\frac{\partial f_{\nu}}{dp^{i}} = \left(\frac{\delta f_{\nu}}{\delta\tau}\right)_{\rm coll} \tag{5}$$

for the neutrino distribution in f_ν in space coordinate x^μ and momentum space coordinate p^i (See [8] for further references). It describes the change in the neutrino distribution along a trajectory with an affine parameter, τ. The right hand side expresses the collision term due to neutrino-matter interactions.

In order to explain the computational structure of neutrino transfer, we write down below the Boltzmann equation in flat space-time under spherical symmetry [3]. The neutrino radiation is represented by a distribution function $f_\nu(t, r, E_\nu, \mu = \cos\theta)$, where E_ν is the neutrino energy and θ the angle of neutrino momentum with respect to the radial coordinate. The time evolution of f_ν obeys the Boltzmann equation,

$$\frac{\partial f_{\nu}}{\partial t} + \mu\frac{\partial f_{\nu}}{\partial r} + \frac{(1-\mu^2)}{r}\frac{\partial f_{\nu}}{\partial \mu} = \left(\frac{\delta f_{\nu}}{\delta t}\right)_{\rm coll} . \tag{6}$$

The collision term contributes to the change of neutrino number, angle and energy (μ, E_ν) by creation, extinction and scattering by the weak interaction with matter. It is important to note that the collision term largely depends on neutrino angle and energy and contains the integral of neutrino distributions by angle and energy variables for scattering processes. It also has contributions from the pair-processes with the distribution functions of neutrinos and anti-neutrinos. The Boltzmann equation, therefore, is an integro-differential and non-linear equation. This fact makes the numerical solution of the Boltzmann equation further computationally demanding.

Equation of State. The equation of state for hot and dense matter is used to provide thermodynamical quantities such as pressure and chemical potentials for hydrodynamics and weak reaction rates. The data table is implemented in the numerical code to cover the wide variety of conditions in supernovae.

2.3 Numerical Scheme

A finite difference form of the set of equations for hydrodynamics and neutrino transfer is solved in our numerical code [7,9]. The discrete ordinate (S_N)

method is adopted to solve the Boltzmann equation through finite differencing the radial and angular advection terms. We solve the Boltzmann equation by the multi-energy group treatment due to the energy dependence of neutrino matter interactions. Since the collision term contains coupling in energy, it is not possible to parallelize in terms of energy.

A fully implicit differencing is adopted for time advance since the equation is stiff with largely different time scales due to the energy dependence. This is advantageous to increase time step, which is constrained by the Courant number in the explicit differencing, to follow the time evolution in a long time scale. However, it is expensive to solve a linear system with large sparse matrix at each step for time advancing. In order to solve the non-linear equations of neutrino-radiation hydrodynamics, the Newton-Raphson iteration is applied to linearize the equations. Therefore, the solution of the linear equations is the most time consuming part of our numerical simulations.

We set the radial coordinate with N_r grid points and angle and energy coordinates of neutrinos with N_{ang} and N_{E_ν} grid points. At each radial point, there are $N_{\text{hyd}} = 11$ hydrodynamical quantities and $N_{E_\nu} \cdot N_{\text{ang}} \cdot N_\nu$ neutrino degrees of freedom, where N_ν is the number of neutrino species. A typical size of spherical simulations at the current supercomputers is $O(100)$ for N_r and $O(10)$ for N_{ang} and N_{E_ν}. Increasing the number of grids to $O(1000)$ for N_r is preferable to obtain necessary resolution of accreting or exploding material on the compact object. Adopting $O(100)$ for N_{ang} and N_{E_ν} is advisable to describe the forward peak in angle at large distance from the center and the sharp drop in energy at the Fermi energy for the neutrino distributions.

2.4 Linear Equation Solver

We focus on the linear equation solver that is called in every step of the time evolution. The linearized equation in the numerical scheme described above results in a block tridiagonal matrix

$$M = \begin{pmatrix} B_1 & C_1 & 0 & & \dots & \\ A_2 & B_2 & C_2 & 0 & & \\ 0 & A_3 & B_3 & C_3 & & \\ \vdots & & \ddots & \ddots & \ddots & 0 \\ & & 0 & A_{n-1} & B_{n-1} & C_{n-1} \\ 0 & \dots & & 0 & A_n & B_n \end{pmatrix}, \tag{7}$$

where $n = N_r$ is the number of radial points. The block matrices A_i, B_i, and C_i at each radial point are dense matrices of rank $N_{\text{max}} = N_{E_\nu} N_{\text{ang}} N_\nu + N_{\text{hyd}}$. This kind of pattern of block matrices often appears in radiation transfer [10] and is similar to the one appears in the multi-dimensional neutrino transfer [3].

We apply an iterative solver algorithm based on the Krylov subspace method. We adopt the BiCGStab algorithm improved by a weighted Jacobi-type preconditioner [11]. As the preconditioning, the following Jacobi iteration is applied as

$$\begin{aligned} x_{k+1} &= \omega[-M_D^{-1}(M_L + M_U)x_k + M_D^{-1}b] + (1-\omega)x_k \\ &= x_k + \omega M_D^{-1}(b - Ax_k), \end{aligned} \tag{8}$$

where ω is the weight parameter, M_D, M_L, M_U respectively represent the block diagonal, lower, and upper parts of M, Eq. (7).

As the preconditioner, slightly modified operation suffices:

$$x_{k+1} = x_k + \omega D^{-1}(b - \tilde{M}x_k), \tag{9}$$

where D and $\tilde{M}$ resembles M_D and M, respectively. Choice of D and $\tilde{M}$ may reduce the arithmetic cost while keeping preconditioning efficient. In this work, however, we set $D = M_D$ and $\tilde{M} = M$ for simplicity of implementation. We note that D^{-1} can be determined as an additional block diagonal matrix before the solver iteration starts. The weight parameter ω is determined as in Ref. [11]. As a preconditioner, the Jacobi iteration (8) is repeatedly applied $N_{\rm Jacobi}$ times before the multiplication of M in the solver algorithm. The value of $N_{\rm Jacobi}$ is tuned by observing the convergence of the solver. In present case, we find $N_{\rm Jacobi} = 25$ being an efficient choice.

For systematic survey of stellar models, it is necessary to keep elapsed time for each time step of evolution within 10 s. Increasing the number of grid points, this condition becomes increasingly hard to satisfy. On massively parallel clusters, this might be achieved by increasing the computing nodes. However, for the spherically symmetric case, such parallelization is inefficient for typical values of N_r unless the degrees of freedom at each radial point, $N_{\rm ang} \cdot N_{E_\nu} \cdot N_\nu$, are also distributed over several nodes, in addition to N_r that the original code does. It would make implementation involved due to the existence of the hydrodynamic degrees of freedom, $N_{\rm hyd}$. An advantage of using GPUs is that one can parallelize the code in units of several radial points while acquiring sufficient computational performance, due to the block structure of M in Eq. (7). In practice, one needs to adjust such parallelization parameters in addition to the numbers of grid points so that the simulation time is kept reasonably short.

3 Implementation

3.1 Offloading Scheme

In this paper, we focus on the iterative linear equation solver as a target of offloading to GPU devices. Our original code is written in Fortran and parallelized in the radial coordinate with standard MPI. As an offloading procedure, we employ OpenACC that is based on the directive-based programming model. One inserts directives in the code to specify the tasks to be executed on devices, and a compiler generates corresponding modules. OpenACC has several attractive features; it does not depend on specific architecture, enables incremental code development for offloading, and can be processed by several compilers that are rapidly increasing efficiency. Since we would need to offload not only the linear solver but also other parts of the simulation code, the simpler in implementation is the better. OpenACC is realized to fulfills this requirement, while OpenMP after version 4.0 may become an alternative. We restrict our implementation in the double precision, which is adopted in the original code.

For offloading the linear equation solver, we first translate the target part of the code to a code in the C language, and then insert the OpenACC directives to the latter. This is partially for future convenience toward employing CUDA or OpenCL as well as application to other architectures. As another reason, we apply a prescription that is effective in the development of a lattice QCD simulation code on GPUs [12]. At the call of the C code, the data layouts of matrices and vectors are changed if necessary. Simultaneously, the arrays of matrices and vectors are extended with zero padding, so that their sizes fit the number of cores in the hardware unit.

In the original code, the radial coordinate is divided into several domains each assigned to a MPI process, and the boundary data are transferred by message passing. In a multiple-GPU case, one GPU device is assigned to one MPI process. The boundary data on the device memory are copied to the host memory, transferred by message passing (while the implementation of MPI may use memory copy), and copied to the device memory. Although this may generally become a bottleneck of computation, in the present case the communication overhead becomes less important as the block matrix size increases, since the transferred data are proportional to $N_{\max}$ while the arithmetic operations are $N_{\max}^2 \times (N_r/N_P)$, where N_P is the number of MPI processes.

On the GPU environment provided by NVIDIA, one can employ CUDA and high performance scientific libraries which are cooperative with OpenACC. We make use of the cuBLAS library, that is a BLAS (Basic Linear Algebra Subprograms) library working on GPUs. This is an alternative to the code implemented by ourselves with OpenACC which is called the 'native' code hereafter. In the following, the performance of these two implementations are compared.

3.2 Implementation Details

In this subsection, our implementation is described in more detail. The version of the OpenACC standard is assumed to be 2.0 or later, to which some of the following prescriptions are only applicable.

Memory Allocation. Since the sizes of matrices and vectors are unchanged during simulation, their areas on the global device memory are allocated at the beginning of execution. Just after the C code allocates the host memory by calling `malloc` function, the corresponding device memory area is allocated by the `enter data create` OpenACC directive. For example, for a global variable `*p_d` of the `double` type that represents a vector, it leads to

```
p_d = (double*)malloc(Nvec*sizeof(double));
#pragma acc enter data create(p_d[0:Nvec])
```

where `Nvec` is the size of vector. Hereafter the device memory area is linked to the pointer `p_d` and can be referred by `p_d`. At the end of the execution, the device memory area is freed by the `exit data delete` directive just before the host memory is freed by calling the `free` function. While the unified memory is available after CUDA 6 on the NVIDIA architecture, our implementation does not adopt it and is based on the standard use of the device memory as above.

Data Transfer. The data transfer between the host and device is managed by the `update` directive in OpenACC. For example,

```
#pragma acc update device(p_d[0:Nvec])
```

transfer the data on the host memory referred by `p_d` to the corresponding area on the device memory. For data locality, the data transfer between the host and device is done only when necessary. This implies that the clause of the OpenACC `data` construct before the parallel region is in general `present` except for variables of small sizes.

Parallel Region. There are two kinds of accelerator compute construct: `kernels` and `parallel`. The former directive entrust the parallelization to the compiler, while the latter is used for manual parallelization. We adopt the latter. In a typical case, a task executed on GPU device is a loop, which is processed with the `loop` construct. We unify the loops that are independently executed by threads by hand before applying the OpenACC directives so that a single loop is divided into thread tasks. All the three levels of the OpenACC model, `gang`, `worker`, and `vector` are thus assigned to a single loop.

Asynchronous Operation. Some of the above operations can be asynchronously performed. For example, bulk part of the matrix-vector multiplication can be performed in parallel with the transfer of the boundary data of the vector. Each directive may have `async` clause with a positive integer *id* that specify the unit of asynchronous operations. The operations specified by *id* are ensured to be finished at the point of the `wait` directive or clause.

Native Code. As a reference, we implement the 'native' code that offloads the matrix-vector multiplication to the device by only using the OpenACC directives. In this implementation, for $y = Ax$, where A is one of M, $M - M_D$, and M_D^{-1}, each component of the vector y is computed by a single thread. This means that each thread executes a loop of size $N_{\max}$.

As an example, let us consider the NVIDIA's GPU that is employed in this paper. Execution of threads are done in units of 'warp' that concurrently access the global device memory. This leads to the so-called 'coalesced access', a fundamental technique in device code tuning, that achieves efficient memory access by storing the data in units of N_{warp}. We store the components of each block matrix A_{ab}, where A is one of A_i, B_i, C_i, and B_i^{-1}, as an array of size $N_{\max}^2$ with a serialized index

$$\mathrm{index}(a,b) = (a\%N_{\mathrm{warp}}) + N_{\mathrm{warp}} \cdot (b + N_{\max} \cdot [a/N_{\mathrm{warp}}]) \tag{10}$$

where $a\%N_{\mathrm{warp}}$ represents a modulo and $[a/N_{\mathrm{warp}}]$ an integer quotient.

Since the above implementation is rather simple and there would be more sophisticated tuning such as loop tiling. At this stage of development, however, we leave further involved tuning for later and incorporate the cuBLAS library as an alternative solution.

Code with cuBLAS. For a dense matrix, a well-tuned library cuBLAS is available on the NVIDIA's environment. We make use of the `cublasDgemv` function that corresponds to `Dgemv` in the original BLAS library. Since originally BLAS has been developed as a Fortran code, the matrix format is column major, *i.e.* column components are continuously stored on the memory. Thus when cuBLAS is employed the matrix data A_{ab} are stored with a serialized index $a + N_{\max} \cdot b$.

Since each block matrix is not enough large to exhaust the cores of GPU device, parallel application of matrix-vector multiplication is essential. This is done by exploiting the CUDA stream that enables asynchronous execution of the operations distributed to the streams. One first setup a `cublasHandle_t` object, a handle of cuBLAS events, through the `cublasCreate` function. The CUDA streams are represented by an array of `cudaStream_t` object which is setup through the `cudaStreamCreate` function. By assigning the cuBLAS handle to one of the CUDA streams, successive cuBLAS functions are executed in that stream.

3.3 Related Works

Although the simulations of core-collapse supernovae are computationally demanding, use of GPUs has been progressed mainly for application to the hydrodynamics. As for the simulation code including the neutrino transport, the VERTEX code was ported to GPUs by employing CUDA of NVIDIA [13]. Since the VERTEX code employs the explicit scheme for the hydrodynamics and Boltzmann equation, its most time consuming part is calculation of the collision term of the Boltzmann equation. One reaction term exhausts almost half the simulation time and thus offloaded to GPUs. On the Kepler architecture, the target kernel is accelerated by factor of 54 which results in 1.8 times acceleration of the whole simulation time compared to that of execution on the host processors.

4 Results

4.1 Numerical Setup

Performance of our code is examined on the following two systems. Both the systems are composed of host processors and NVIDIA's GPUs, as summarized

Table 1. Machines used for performance measurement

	Machine-1	Machine-2
Host processor	Intel Xeon E52643v3 x2	IBM Power8 x2
Host cores	12	20
Peak performance (double)	44.8 GFlops/core	22.9 GFlops/core
Host memory [GB]	64	512
Host-device connection	PCI-E 3.0 x16	NVLink
GPU	NVIDIA Tesla K40 x2	NVIDIA Tesla P100 x4
FP64/FP32 CUDA cores/GPU	960/2880	1792/3584
Peak GFlops (double/float)	1430/4290	4700/9300
Memory size [GB]	12	16
Memory B/W [GB/s]	288	720

Table 2. Parameter sets for measurement. The value of memory requirement is for the matrices and determined as $N_r \cdot N_{\max}^2 \cdot 4 \cdot 8$ Byte.

	Set-1	Set-2	Set-3	Set-4
N_r	256	256	256	256
N_{E_ν}	14	16	24	32
$N_{\rm ang}$	6	8	12	16
N_ν	4	4	4	4
$N_{\max}$	347	523	1163	2059
Memory requirement [GB]	1	2	9.3	32

in Table 1. The machine-1 is composed of two Intel Xeon processors and two K40 GPUs (Kepler architecture) connected with PCIe Gen3 of 16 lanes. The machine-2 is composed of two POWER 8 processors with four P100 GPUs (Pascal architecture) connected by NVLink. On both the systems, we use the PGI compiler with OpenMPI. Since currently popular large scale systems including GPUs typically adopt similar structure as a node, we consider they provide typical examples of practical environment.

The size of block matrices A_i, B_i, and C_i are determined by the choice of N_{E_ν} and $N_{\rm ang}$. The parameter N_ν is the number of neutrino species and fixed to 4 (electron-type neutrino, muon-type neutrino, and their antiparticles). The rank of block matrices is $N_{\max} = N_{E_\nu} \cdot N_{\rm ang} \cdot N_\nu + H_{\rm hyd}$. Since they are dense matrices, the required memory size is proportional to $N_{\max}^2$. The total size of required memory space is also proportional to N_r. In Table 2, we display the sets of parameters examined in this work. The value of memory requirement for the matrices (including M_D^{-1}) is determined as $N_r \cdot N_{\max}^2 \cdot 4 \cdot 8$ Byte. Set-1 corresponds to the parameters currently employed in practice. From Set-1 to Set-4, the rank of the block matrices are gradually increased. Set-4 is already too large for single

GPU or two GPUs to allocate the device memory space, so that it is measured only on the machine-2 with 4 GPU devices.

4.2 Performance Results

We first examine the performance of matrix-vector multiplication. As explained in the description of solver, three matrices are applied to vectors: the full block tridiagonal matrix M, the subdiagonal matrix $M - M_D = M_L + M_U$, and the inverse of block diagonal matrix M_D^{-1}. Note that the block matrices of M_D^{-1} are determined on the host processor before offloading to the devices. To single out the matrix multiplication performance, we start with examination of M_D^{-1}, since it includes no inter-device communication.

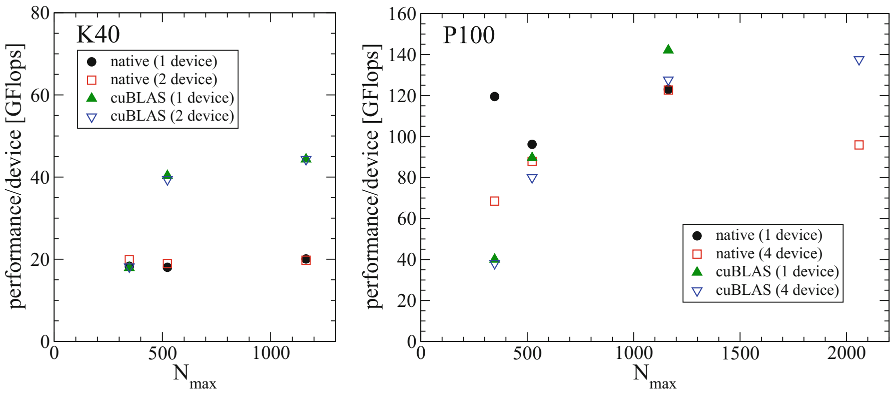

Fig. 1. The performance of the multiplication of M_D^{-1}. The left and right panels show the results on the machine-1 (with K40) and 2 (P100), respectively. On each machine, the native and cuBLAS codes are executed using single and multiple GPU devices.

The performance is measured on the machine-1 and 2, which are respectively denoted by K40 and P100 in the figures. On each system, the performance is measured with single GPU (if possible) and multiple GPUs using the native code and the code with cuBLAS library. For the latter, the number of CUDA stream, N_{stream}, is a tunable parameter. At each combination of the parameter set, the machine, and the number of devices, we observe the dependence of performance on N_{stream} and adopt the value that provides the best performance for $M - M_D$ multiplication, since it is frequently called during the Jacobi iteration and costs more than M_D^{-1}. We measure the elapsed time for 200 times of matrix-vector multiplication and obtain the sustained performance based on the count of floating point operations in the code.

Multiplication of M_D^{-1}. Figure 1 displays the performance of M_D^{-1} multiplications against the rank of the block matrices. The left and right panels show the results on machine-1 and 2, respectively. For the Set-4 parameter set, due to the memory restriction, only the multi-device results on the machine-2 are available. Both the performances of the native and cuBLAS codes increase as $N_{\max}$ and seem to saturate at $N_{\max} = 1163$. In the case of native code, the performance of the multi-device is worse, in particular for small $N_{\max}$ region. A reason is considered that the number of threads is not sufficiently large to exhaust the large number of cores. While the cuBLAS code shows lower or similar performance to that of the native code at $N_{\max} = 347$, it quickly conquers for larger $N_{\max}$. This is an appearance of more sophisticated use of cores in the cuBLAS library for a sufficiently large matrix.

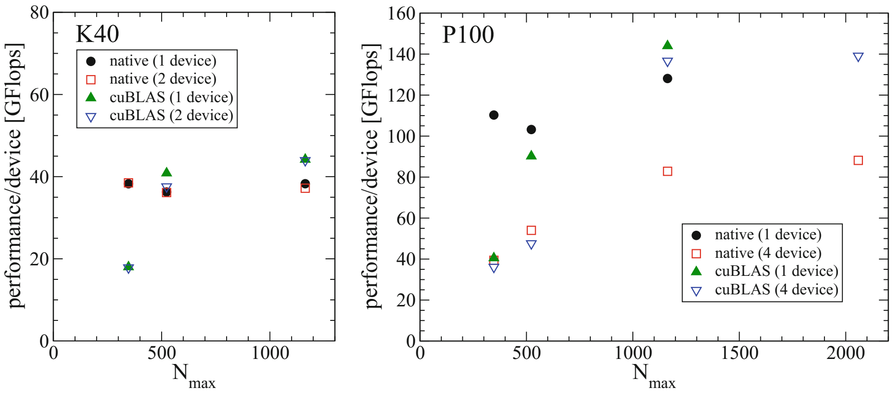

Fig. 2. The performance of the multiplication of $M - M_D$. The condition of measurement is the same as in Fig. 1.

Multiplication of $M - M_D$ and M. We next examine the performance of matrices that require the inter-device communication. Figures 2 and 3 display the performance of multiplications of $M - M_D$ and M to vectors, respectively. As a general tendency, native code shows better performance than those for M_D^{-1}. This is explained by that the two or three block matrices are simultaneously multiplied to a vector in each thread. In contrast the cuBLAS code exhibits similar performance as the M_D^{-1} case, since the cuBLAS functions are called in units of block matrices. In both the cases, the overhead of the boundary copy is small, in particular for larger $N_{\max}$ region. Indeed, performance of a code with synchronous boundary data copy is almost the same as that of the asynchronous code which provided the results in the figures.

Scaling with N_r. Finally we examine the scaling with N_r. To achieve our preferable resolution, it is mandatory to increase the value of N_r up to $N_r = 1024$. Considering the computational resource and balance of N_r and $M_{\max}$, practical choice of parameters is $N_{E_\nu} = 24$ and $N_{\rm ang} = 12$. We thus observe N_r dependence of the performance on the machine-2 with four GPU devices. The result at $N_r = 1024$ is almost consistent with the results of the same $N_{\max}$ and $N_r = 256$ (*i.e.* Set-2) on the single device. This indicates that the present code shows good weak scaling behavior in N_r at least within the node.

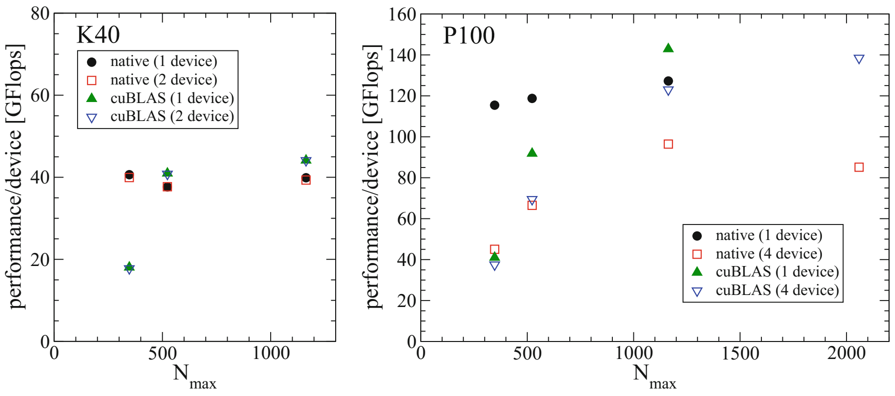

Fig. 3. The performance of the multiplication of M. The condition of measurement is the same as in Fig. 1.

Impact on Simulation Time. Before considering further elaborated tuning, let us examine the impact of the above offloading code on the whole numerical simulation. At run time, the code first read several files to set the initial condition of degrees of freedom together with the table data of the equation of state, and then the time evolution step begins. At each step of time evolution, the contributions of hydrodynamics, collision term, and advection terms to the evolution matrix (7) are computed in order. Among them, calculation of the collision term is the most time consuming part. Other parts are almost negligible compared to the linear solver and the collision term. In solving the linear equation, first the weight parameter of the Jacobi iteration and M_D^{-1} are determined (denoted by "setup"), before the application of an iterative solver algorithm. Only the iterative solver is offloaded in the present paper.

Table 3 shows the elapsed time of each ingredient in one Newton-Raphson update that is repeated a few times during each evolution time step. The data on GPU devices are measured with the code with cuBLAS library. To examine the scaling behavior of the non-offloading code against the number of MPI processes N_p on the host processors, we also measure the cases with N_p larger than the number of GPU devices. Within our computational resources, the code shows good scaling behavior for the MPI parallelization. Although the code for the host processors has not been well optimized for the current architectures, it seems difficult to achieve the elapsed time less than 10 s unless the number of MPI processes is substantially increased. On the other hand, the results of the offloaded iterative solver displays illustrative speedup toward the desired elapsed time. It implies the possibility to keep the elapsed time for each time step within several seconds if other parts, the computation of collision term and the setup of linear solver, are also accelerated. Thus offloading these calculation to GPU devices would be an urgent subject rather than further optimization of the iterative solver.

Table 3. Impact of offloading the iterative solver on the simulation time. The data on GPU devices are measured with the code with cuBLAS library.

	# process	Collision [sec]	Matrix inversion	
			Setup [sec]	Solver [sec]
Set-1, machine-1	8	5.0	5.1	110.9
Set-1, machine-1 (with GPU)	1	40.4	40.4	865.6 5.3
Set-2, machine-2	16	8.6	11.8	170.4
Set-2, machine-2 (with GPU)	4	35.0	44.9	627.0 0.73

5 Conclusion

In this work, we examined feasibility of the architectures possessing GPU devices for numerical simulations of core-collapse supernovae. The most time consuming part of the implicit scheme is a linear equation solver so that its acceleration is essential for this type of simulation. We develop a code to offload this part to GPUs using OpenACC as well as the cuBLAS library provided by NVIDIA. While there is a room to improve the performance, the elapsed time for the linear equation solver is sufficiently reduced so that it is no longer a bottleneck. The result is encouraging to extend the offloading to other time consuming parts of the simulation. Since the OpenACC is also applicable to a Fortran code, acceleration of other parts would be possible in a similar manner. An encouraging result for the VERTEX code was reported for computation of the collision term of the Boltzmann equation in Ref. [13].

Toward multi-dimensional simulations of core-collapse supernovae, there are several practical issues to be examined. As the dimension increases, the problem size drastically increases and a large scale system is inevitably required. In such a system, the inter-node communication becomes increasingly important. More elaborated management of arithmetic operations and boundary data transfer would be required. Extending our implementation to multi-dimensional simulation code is now underway.

Acknowledgment. The authors are grateful to A. Imakura, H. Okawa, T. Takiwaki and S. Yamada for HPC researches on core-collapse supernovae. For providing high performance computing resources, Computing Research Center, KEK, Research Center for Nuclear Physics, Osaka University, Yukawa Institute of Theoretical Physics, Kyoto University, and Information Technology Center, University of Tokyo are acknowledged. This work was supported by Grant-in-Aid for the Scientific Research from the Ministry of Education, Culture, Sports, Science and Technology (MEXT), Japan (26104006, 15K05093, 17H06357, 17H06365) and HPCI Strategic Program of Japanese MEXT and research programs at K-computer of the RIKEN AICS and Post-K project (Project ID: hp 170230, 170031, 170304, 180179, 180111).

References

1. Kotake, K., Sumiyoshi, K., Yamada, S., Takiwaki, T., Kuroda, T., Suwa, Y., Nagakura, H.: Core-collapse supernovae as supercomputing science: a status report toward 6D simulations with exact Boltzmann neutrino transport in full general relativity. Prog. Theor. Exp. Phys. **2012**, Article no. 01A301 (2012). https://doi.org/10.1093/ptep/pts009
2. Janka, H.-T., Melson, T., Summa, A.: Physics of core-collapse supernovae in three dimensions: a sneak preview. Ann. Rev. Nucl. Part. Sci. **66**, 341–375 (2016). https://doi.org/10.1146/annurev-nucl-102115-044747
3. Sumiyoshi, K., Yamada, S.: Neutrino transfer in three dimensions for core-collapse supernovae. I. Static configurations. Astrophys. J. Suppl. **199**, 17 (2012). https://doi.org/10.1088/0067-0049/199/1/17
4. Liebendoerfer, M., Mezzacappa, A., Thielemann, F.-K., Messer, O.E., Hix, W.R., Bruenn, S.W.: Probing the gravitational well: No supernova explosion in spherical symmetry with general relativistic Boltzmann neutrino transport. Phys. Rev. D **63**, Article no. 103004 (2001). https://doi.org/10.1103/PhysRevD.63.103004
5. Sumiyoshi, K., Yamada, S., Suzuki, H., Shen, H., Chiba, S., Toki, H.: Postbounce evolution of core-collapse supernovae: long-term effects of the equation of state. Astrophys. J. **629**, 922–932 (2005). https://doi.org/10.1086/431788
6. Nagakura, H., Iwakami, W., Furusawa, S., Okawa, H., Harada, A., Sumiyoshi, K., Yamada, S., Matsufuru, H., Imakura, A.: Simulations of core-collapse supernovae in spatial axisymmetry with full Boltzmann neutrino transport. Astrophys. J. **854**, 136 (2018). https://doi.org/10.3847/1538-4357/aaac29
7. Yamada, S.: An implicit Lagrangian code for spherically symmetric general relativistic hydrodynamics with an approximate Riemann solver. Astrophys. J. **475**, 720–739 (1997). https://doi.org/10.1086/303548
8. Shibata, M., Nagakura, H., Sekiguchi, Y., Yamada, S.: Conservative form of Boltzmann's equation in general relativity. Phys. Rev. D **89**, Article no. 084073 (2014). https://doi.org/10.1103/PhysRevD.89.084073
9. Yamada, S., Janka, H.-T., Suzuki, H.: Neutrino transport in type II supernovae: Boltzmann solver vs. Monte Carlo method. Astron. Astrophys. **344**, 1468–1470 (1999). arXiv:astro-ph/9809009
10. Sumiyoshi, K., Ebisuzaki, T.: Performance of parallel solution of a block-tridiagonal linear system on Fujitsu VPP500. Parallel Comput. **24**, 287–304 (1998). https://doi.org/10.1016/S0167-8191(98)00007-6
11. Imakura, A., Sakurai, T., Sumiyoshi, K., Matsufuru, H.: A parameter optimization technique for a weighted Jacobi-type preconditioner. JSIAM Lett. **4**, 41–44 (2012). https://doi.org/10.14495/jsiaml.4.41
12. Matsufuru, H., et al.: OpenCL vs OpenACC: lessons from development of lattice QCD simulation code. Procedia Comput. Sci. **51**, 1313 (2015). https://doi.org/10.1016/j.procs.2015.05.316
13. Dannert, T., Marek, A., Rampp, M.: Porting Large HPC Applications to GPU Clusters: The Codes GENE and VERTEX. Advances in Parallel Computing, vol. 25, pp. 305–314 (2014). https://doi.org/10.3233/978-1-61499-381-0-305