Performance Portability Analysis for Real-Time Simulations of Smoke Propagation Using OpenACC

Anne Küsters[1(✉)], Sandra Wienke[2,3], and Lukas Arnold[1]

[1] JSC, Forschungszentrum Jülich GmbH, Wilhelm-Johnen-Straße, 52428 Jülich, Germany
a.kuesters@fz-juelich.de

[2] IT Center, RWTH Aachen University, Seffenter Weg 23, 52074 Aachen, Germany

[3] JARA-HPC, 52074 Aachen, Germany

Abstract. Real-time simulations of smoke propagation during fires in complex geometries challenge engineers, physicists, mathematicians and computer scientists due to the complexity of fluid dynamics and the large number of involved physical and chemical processes. Recently, several application scenarios emerged that require real-time predictions during an incident to support the rescue teams. Therefore, we develop the CFD-based simulation software JuROr aiming to run in real-time by leveraging parallel computer architectures like CPUs and GPUs. For that, we parallelize the code with OpenACC directives that promise maintenance of a single source base by delegating some architecture-agnostic optimizations to the compiler. We investigate the performance portability of JuROr using PGI's OpenACC implementation across four Intel CPUs and three NVIDIA GPUs. We present the achieved performance shares as part of a roofline model where we focus on traditionally-computed arithmetic code intensities, as well as on a measurement approach based on performance counters.

Keywords: Parallel CFD applications · Fire safety engineering · GPU computing · OpenACC · Performance portability · Roofline model

1 Introduction

In almost all underground stations in Germany, the equipment for smoke extraction remains rare to find. To support effective firefighting measures and tactics, the long-term goal is to develop a decision making tool for firefighters in cases of complex geometries where air and smoke flows are both complex and hard

The original version of this chapter was revised: The ORCIDs of second and third authors have been corrected. The erratum to this chapter is available at https://doi.org/10.1007/978-3-319-67630-2_54 to predict. Therefore, concepts must adapt to the current situation dynamically and scenario-based.

During the last decade, computational fluid dynamics (CFD) has gained much attention in fire safety engineering by simulating smoke propagation. However, currently, users of commercial or open source smoke simulation tools widely apply these methods to simplified geometries that do not fit the regulatory prescription of the Pattern Building Code (e.g. enabling escape, rescue and effective fire fighting measures). Thus, these complex geometries need individual evaluation. Furthermore, it remains a challenge for them to meet the crucial constraint of simulating the smoke propagation in real time or less.

To simulate smoke propagation in complex 3D geometries, we develop a C++-based CFD solver, called 'JuROr' (Jülich's Real-Time Simulation within Orpheus). The goal of the Orpheus project funded by the 'Federal Ministry of Education and Research' (BMBF) is the improvement of personal safety in underground stations in case of fire. Further information can be found in [1].

We parallelize the CPU-based solver JuROr using the directive-based programming model OpenACC and leverage the compute power of CPUs and GPUs to enable real-time simulations. One advantage of using OpenACC over a low-level accelerator model is the delegation of the responsibility of producing performance-portable code to the compiler. Here, we investigate the performance portability of PGI's OpenACC implementation across various hardware architectures: NVIDIA Kepler and Pascal GPUs, and Intel Xeon Sandy Bridge, Ivy Bridge, Haswell and Broadwell CPUs (using PGI's multicore target). For that, we build roofline models [2] for the different architectures, i.e., we model performance limiters such as Flop/s and memory bandwidth on base of the application's arithmetic intensity. Then, we present performance portability of the real-world code JuROr as percentage of sustainable peak performance.

Thus, our main contributions are:

- A CFD solver and its parallelization with OpenACC to enable the prediction of smoke propagation in complex rooms
- Analysis of its performance portability using the roofline model based on manually-computed arithmetic intensity vs. measured intensity, and hardware performance counters
- Investigation of various hardware architectures: three NVIDIA GPUs, four Intel CPUs

The paper is structured as follows: Sect. 2 presents related work regarding CFD solvers utilizing GPUs as well as performance portability analysis of OpenACC codes. We introduce JuROr's CFD methods solving weakly compressible Navier-Stokes equations for a turbulent flow in Sect. 3 and its parallelization using OpenACC in Sect. 4. In Sect. 5, we describe our approach for modeling and measuring performance. We present our results on performance portability in Sect. 6. Finally, we conclude and give a short outlook in Sect. 7.

2 Related Work

So far, no possible solutions exist in the field of flow simulations for smoke propagation in real-time using CFD and covering complex rooms while taking sensor data into account. Yet, interest in producing real-time predictions, like those investigated in the *FireGrid* project [3], already exists. However, the utilized fire simulation model in FireGrid is a zone model, which splits the domain of interest into very few zones (cf. [4]). Properties like temperature or smoke density are computed via a set of coupled ordinary differential equations (ODEs) and thus only allow for very crude approximations and applications on simple geometries.

Instead of using a zone model, Glimberg et al. studied the governing mathematical models of CFD which describe the smoke propagation sufficiently (cf. [5]). In their work, GPUs were employed to solve the governing equations in highly simplified geometries using a fractional step method. This approach resulted in a solution within less than a minute of runtime for ten seconds simulation time. For comparison, the simulation took more than one hour on a CPU. However, this approach did not include the coupling of sensor data.

On the basis of sensor data, Daniel and Rein implemented *The Fire Navigator* forecasting the spread of building fires (cf. [6]). Using the techniques of a cellular automata building fire model (instead of CFD), they employed sensor data assimilation, inverse modeling and genetic algorithm techniques. With this approach the governing parameters of a fire, such as the flame spread rate, the smoke ceiling jet velocity and the outbreak location and time, can be indirectly uncovered and then used to produce real-time as well as forecast maps of the flame spread and smoke propagation. Therewith, the *Fire Navigator* achieves positive lead times of several minutes meaning the predictions are actually forecasts (without the usage of GPUs). Nonetheless, cellular automata simulations simplify the problem and do not produce results as accurate as computational fluid dynamics.

Instead, in JuROr we aim to predict smoke propagation in complex geometries deploying computational fluid dynamics and taking sensor data into account in future works.

By leveraging parallel processing power of GPUs and CPUs, JuROr tackles the real-time constraints for complex geometries. Its OpenACC parallelization shall enable good performance across various kinds of clients' hardware architectures. Although the architecture-specific assembler optimization by OpenACC compilers ease the maintenance of a single code base (e.g. over using OpenCL [7]), the performance portability of OpenACC implementations have been scarcely studied so far. While OpenACC performance comparisons across different architectures have been targeted in research, they have mostly been conducted by taking absolute numbers such as the application's runtime, floating point operations per second or speedup over CPU runs.

For example, Lopez et al. [8] show memory bandwidth or speedup numbers for a Jacobi and n-body kernel for different OpenACC implementations (PGI, Cray) on NVIDIA Kepler GPUs. They failed on using OpenACC on multicore CPUs. Sabne et al. [9] evaluated the performance by showing speedup numbers based on OpenARC's OpenACC implementation on NVIDIA GPUs, AMD GPUs and Intel Xeon Phi coprocessors using 12 kernels. The hydrodynamic mini-app CloverLeaf [10] has been tested on NVIDIA GPUs, Intel Xeon Phi Coprocessors, one AMD APP and different CPUs using the vendor OpenACC implementations from CAPS, PGI and Cray. For real-world codes, Nicolini et al. [11] present runtimes of an aeroacoustic simulation software package using PGI's OpenACC implementation for NVIDIA Kepler GPUs and Intel CPUs. Calore et al. [12] investigate a lattice Boltzmann application also using PGI's OpenACC implementation on NVIDIA GPUs, AMD GPU and Intel CPU.

However, performance portability investigations should not only consider absolute numbers, but need to account for the hardware's and application's characteristics. For that, some studies [9,10,12] compare their gained OpenACC performance to hand-tuned low-level implementations written in CUDA or OpenCL, or to libraries like MKL or CUBLAS. As a percentage of peak performance, Lopez et al. [8] present their DAXPY and DGEMV kernels. For two non-trivial kernels, Calore et al. [12] show an OpenACC efficiency of 54% to 70% of peak across different architectures for memory-bound code, while compute-bound code achieves 14% to 24% efficiency.

Modeling OpenACC performance using a roofline model has only been conducted by Wang et al. [13] who base their model on STREAM and Flop/s measurements and apply CAPS' OpenACC implementation to NVIDIA GPUs and Intel Xeon Phi coprocessors. However, they only examine basic kernels from the EPCC OpenACC Benchmark Suite. There, they get up to 82% of sustained performance while most kernels achieve about 10%. In contrast, we do not only focus on absolute performance, but especially apply the roofline model to the real-world code JuROr.

Nomenclature

C_S	Smagorinsky constant	β	thermal extension coefficient
$\mathbf{f}$	force ($\frac{kg \cdot m}{s^2}$)	κ	thermal conductivity ($\frac{W}{m \cdot K}$)
$\mathbf{g}$	gravitational force ($\frac{kg \cdot m}{s^2}$)	μ	dynamic viscosity ($\frac{kg}{m \cdot s}$)
N_x	number of cells in x-direction	ν	kinematic viscosity ($\frac{m^2}{s}$)
N_y	number of cells in y-direction		
p	pressure ($\frac{kg}{m \cdot s^2}$)	ρ	density($\frac{kg}{m^3}$)
S	stress tensor	$\overline{(\cdot)}$	filtered quantity
S_T	source term ($\frac{kg \cdot m}{s^2}$)	$(\cdot)_{eff}$	effective quantity
T	temperature (K)	Δ_f	filter width
T_0	ambient temperature (K)	$(\cdot)_{mol}$	molecular quantity
t	time (s)	$(\cdot)_t$	turbulent quantity
$\mathbf{u}$	velocity ($\frac{m}{s}$)	Δt	size of time step (s)
$\mathbf{u}_0$	velocity at time $t = 0$ ($\frac{m}{s}$)	Δx	grid size in x-direction (m)
$\mathbf{x}$	point in space $(x, y)^\top (m)$	Δy	grid size in y-direction (m)

3 Numerical Methods of JuROr

To simulate the transport of hot smoke, we first introduce the governing equations which mathematically describe the physics of smoke propagating. Then, we describe the numerical methods approximating the solution of those equations.

3.1 Governing Equations

Smoke propagation can be described with the weakly compressible Navier-Stokes equations (1) and (2) for a turbulent gas with velocity $\mathbf{u}$, pressure p and temperature T as well as no-slip boundary conditions ($\mathbf{u} = \mathbf{0}, \nabla p = 0$) at the walls

$$\partial_t \mathbf{u} + (\mathbf{u} \cdot \nabla)\,\mathbf{u} - \nu \nabla^2 \mathbf{u} + \frac{1}{\rho} \nabla p = \mathbf{f}(T) \tag{1}$$

$$\nabla \cdot \mathbf{u} = 0 \tag{2}$$

$$\partial_t T + (\mathbf{u} \cdot \nabla)\,T - \kappa \nabla^2 T = S_T. \tag{3}$$

Here, weakly compressible means that the density is dependent on the temperature $\rho = \rho(T)$ wherefore the energy equation (3) has to be solved. The force density is described by $\mathbf{f}(T) = -\beta(T - T_0)\mathbf{g}$, where β represents the thermal extension coefficient, T_0 is the ambient temperature and $\mathbf{g}$ is the gravitational force.

For the sake of computing time, we neglect pyrolysis, combustion, and heating/ cooling of surrounding walls and therefore, we focus only on the transport of hot smoke. For this case, we only take the fire far field into account and therefore, we simply consider pyrolysis and combustion by prescribing a mass and heat source.

3.2 Numerical Approach

To solve the governing equations, we take a finite difference (FD) approach on a regular grid. In space, we use central finite differences of 2^{nd} order and in time backwards differencing of 1^{st} order.

Therewith, we implement a fractional step method which follows the scheme outlined in Glimberg's work [5]:

$$\partial_t \mathbf{u}_1 = -\left(\mathbf{u}_1 \cdot \nabla\right) \mathbf{u}_1 \tag{4}$$

$$\partial_t \mathbf{u}_2 = \nu \nabla^2 \mathbf{u}_2 \tag{5}$$

$$\partial_t \mathbf{u}_3 = \mathbf{f}(T) \tag{6}$$

$$\partial_t \mathbf{u}_4 = -\tfrac{1}{\rho} \nabla p. \tag{7}$$

Advection via a Semi-Lagrangian Approach. The idea is to trace back velocities in time to find the current velocities since they do not change along streamlines according to the method of characteristics. Thus, we calculate the starting point from the current position (back tracing) $\mathbf{x}_d = \mathbf{x} - \Delta t \mathbf{u}_0$ to calculate the current velocity in (4) with bilinear interpolation $\mathbf{u}_1 = \mathbf{u}_0\left(\mathbf{x}_d(-\Delta t, \mathbf{x})\right)$. This method is stable in time since it is true that $\max(|\mathbf{u}_1|) \leq \max(|\mathbf{u}_0|)$ holds for all times.

Diffusion with an Implicit Jacobi Method. After applying backwards differencing in time to (5)

$$\frac{\mathbf{u}_2 - \mathbf{u}_1}{\Delta t} = \nu \nabla^2 \mathbf{u}_2,$$

we get a linear system of equations $\left(\mathbf{I} - \Delta t \nu \nabla^2\right)\mathbf{u}_2 = \mathbf{u}_1$ which is solved with Jacobi's method

$$x_i^{(k+1)} = \frac{1}{a_{ii}}\left(b_i - \sum_{j \neq i} a_{ij} x_j^{(k)}\right)$$

for $\mathbf{A} = \left(\mathbf{I} - \Delta t \nu \nabla^2\right)$, $\mathbf{x} = \mathbf{u}_2$ and $\mathbf{b} = \mathbf{u}_1$.

External Forces via Euler Scheme. With an explicit Euler scheme in time, we get a discretized version of Eq. (6) $\mathbf{u}_3 = \mathbf{u}_2 + \Delta t \mathbf{f}$. In order to update the temperature T, we need to additionally solve the energy equation

$$\partial_t T + (\mathbf{u} \cdot \nabla) T - \kappa \nabla^2 T = S_T,$$

where κ characterizes the thermal diffusion coefficient and S_T a temperature source term. Since the energy equation again describes advection and diffusion with a source term, all of the above methods can be applied here.

Pressure Equation with a Geometric Multigrid Method. After backwards differencing the Laplace equation (7) in time to get $\mathbf{u}_4 = \mathbf{u}_3 - \frac{\Delta t}{\rho}\nabla p$, we deploy the incompressibility of $\mathbf{u}_4$ yielding

$$0 = \nabla \cdot \mathbf{u}_4 = \nabla \cdot \mathbf{u}_3 - \frac{\Delta t}{\rho}\nabla^2 p.$$

Now, we solve the pressure-poisson equation $\nabla^2 p = \nabla \cdot \mathbf{u}_3$ with the multigrid method reusing the Jacobian method in the relaxation phases.

Incompressibility Through Projection. We establish incompressibility through orthogonal projection using the Helmholtz-Hodge decomposition by Chorin [14]. Therefore, we define a linear orthogonal projection of $\mathbf{u}$ onto P via $P(\mathbf{u}_3) = \mathbf{u}_4$ such that $\mathbf{u}_3 = P(\mathbf{u}_3) + \nabla p$ with $P(\nabla p) = 0$ to get $\mathbf{u}_4 = \mathbf{u}_3 - \nabla p$.

Turbulence with an Implicit, Constant Smagorinsky-Lilly Large Eddy Simulation (LES). We are solving the LES equations for the spatially filtered velocity $\bar{\mathbf{u}}$ and temperature $\bar{T}$ with filter width $\Delta_f = (\Delta x \Delta y)^{\frac{1}{2}}$ and an effective viscosity of

$$\nu_{eff} = \frac{\mu_{eff}}{\bar{\rho}} = \frac{\mu_{mol} + \mu_t}{\bar{\rho}},$$

where $\mu_t = \bar{\rho} C_S^2 \Delta_f^2 |\bar{S}|$ with Smagorinsky constant C_S [15] (commonly set to $C_S = 0.2$) and the norm of the filtered stress tensor $|\bar{S}| = \sqrt{2\bar{S}_{ij}\bar{S}_{ij}}$, where $\bar{S}_{ij} = \frac{1}{2}\left(\partial_{x_j}\bar{u}_i + \partial_{x_i}\bar{u}_j\right)$.

4 Parallelization with OpenACC

The OpenACC parallelization of the JuROr software is based on a serial runtime profile that can be seen in Fig. 1. The diffusion and pressure methods (depicted in blue) take 50% to 65% (in sum) of the runtime on an Intel Sandy Bridge CPU. The shares for diffusion and pressure highly depend on the problem size, i.e. we get 65% for $N_x = N_y = 512$ and 50% for $N_x = N_y = 2048$. Within the two methods of diffusion and pressure, the 5-point Jacobian stencil operation takes 20% to 80% of the serial runtime when measured by Intel VTune's hotspot analysis. Thus, the Jacobian stencil describes the hotspot of the CPU code and has been parallelized first using OpenACC's `kernels` and `data` regions.

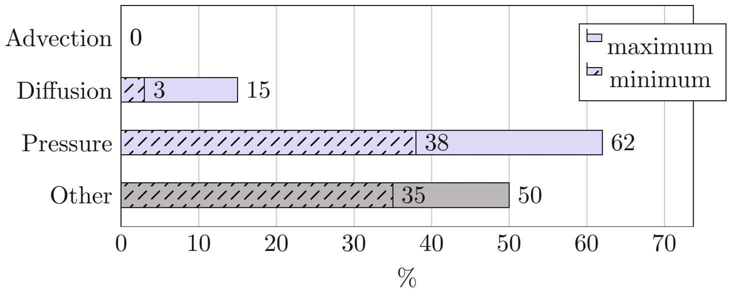


Fig. 1. Share of runtime for different JuROr kernels on one Intel Sandy Bridge core with $N_x = N_y \in \{512, \ldots, 4096\}$ (Color figure online)

After parallelizing the hotspot, all outstanding parallelizable methods (e.g., advection, pressure, boundary conditions) were also ported to the GPU. While we marked all applicable loop nests as parallelizable `independent loops`, we did not specify a certain loop schedule in order to leave it up to the compiler to choose an appropriate loop schedule for the corresponding target architecture. This is an important step in reaching performance portability. Furthermore, we maximized the parallelism across loops by merging smaller loops into one kernel. To reduce the kernel launch latency, we enabled pipelining by `async`hronous kernel launching from the CPU. Data management optimizations include the minimization of data transfers. For example, the access to C++ member attributes in parallelized subroutines caused unnecessary CPU-to-GPU and GPU-to-CPU transfers which were avoided by introducing local variables. In the OpenACC CPU versions, all data transfers are automatically ignored by the compiler so that we can use the same code base for GPU and CPU execution.

For all performance measurements, we run a benchmark test case of JuROr in double precision. This test case describes a $2D$ Navier-Stokes equation comprising advection, diffusion and pressure (without turbulence or external forces) in a $[0, 2\pi]^2$ cube, which are solved using the methods in Sect. 3. The underlying uniform grid varies from being coarse (with 8×8 cells) to very fine (with 4096×4096 cells), where each single cell stores the local values of the variables $\mathbf{u}$ and p. Additionally, we introduced ghost cells (two in each direction) to handle the boundary conditions properly. Thus, the memory size of one matrix of our biggest data set comprises roughly $4098 \times 4098 \times 8 \approx 135$ MB. While this data set fits into our CPU main memory and GPU global memory, it exceeds the CPU and GPU cache sizes.

5 Roofline Model

To investigate the performance portability of our JuROr parallelization using the PGI compiler, we setup a roofline performance model that allows comparison of achieved performance as percentage share of (sustainable) peak performance.

The roofline model [2] builds upon peak floating point performance and sustainable memory bandwidth. It assumes that computation and communication can be completely overlapped and takes only the slowest data path into account. Based on this assumptions, we build our roofline model for seven different hardware architectures that are listed in Table 1: four Intel CPUs and three NVIDIA GPUs. It is noteworthy that we use either one CPU socket or one GPU chip of the given hardware, and do not consider GPU-CPU hybrid computations for now. Correspondingly, we only model performance bounds for either the CPU or GPU chip, even though the host of a GPU-based system actually adds theoretical peak performance to the GPU performance limiters. The latter would also require a corresponding two-device roofline model with inclusion of data transfers which is out of the scope of this paper. Further, we compute the theoretical arithmetic intensity (A.I.) of JuROr and compare it to the measured value by using performance counters. We present the corresponding approaches in the following subsections.

For clarification, we will use the following terminology for our performance numbers:

– *theoretical*: values defined in or computed from technical hardware specifications or from manual code investigations
– *sustainable*: upper performance values that might be obtained in real world usually using benchmarks
– *measured/ achieved*: actual measured performance values with real codes on real hardware.

5.1 Peak Floating-Point Performance and Sustainable Memory Bandwidth

To get the architectural performance limiters, we compute the peak double-precision floating-point performance and measure the bandwidth using (micro) benchmarks.

Calculating Flop/s numbers, we need to consider that most architectures nowadays provide boosting capabilities of the clock frequency that are applied if thermal processor conditions allow it. Since this is difficult to track, we disable auto boosting where possible and base our Flop/s computations on the base operational frequency of the CPU or GPU given in Table 1. This approach is in line with the reporting rules of the *Rpeak* value of the Top 500 list [17].

Regarding the memory bandwidth measurement, it holds that achievable memory bandwidth can be significantly lower than the theoretical peak bandwidth. This is especially true for systems that employ error correcting code (ECC) such as our given architectures do. Therefore, we use benchmarks to obtain the sustainable memory bandwidth. For the GPU systems, we take the CUDA version of the GPU-STREAM benchmark [19,20] and evaluate the bandwidth of the triad kernel. We verify our measurements using the SHOC benchmark [21] as well as comparing them with the published results on the GPU-STREAM website (where possible). For the CPU systems, we take the triad results of McCalpin's OpenMP STREAM benchmark [18] using the Intel Compiler with the flag `-qopt-streaming-stores=always`. We verify these results using Intel VTune's memory access analysis that automatically evaluates the local DRAM single-package bandwidth using a (not further specified) micro benchmark. This micro benchmark delivers slightly higher bandwidth numbers

Table 1. Used hardware architectures and compilers

Name	Hardware	Used	Compiler and Flags
BDW	2-socket Intel Xeon Broadwell E5-2650 v4 @2.20 GHz, 2×12 cores	1 socket	PGI 16.10 -ta=multicore
HSW	2-socket Intel Xeon Haswell E5-2680 v3 @2.50 GHz, 2×12 cores	1 socket	PGI 16.1 -ta=multicore
SNB	2-socket Intel Xeon Sandy Bridge E5-2650 0 @2.00 GHz, 2×8 cores	1 socket	PGI 16.1 -ta=multicore
IVB	2-socket Intel Xeon Ivy Bridge E5-2640 v2 @2.00 GHz, 2×8 cores	1 socket	PGI 16.1 -ta=multicore
P100	NVIDIA Pascal P100 SMX2 GPU, 1328 MHz, 16 GB, autoboost off, ECC on, BDW host	1 GPU	PGI 16.10 -ta=tesla:cc60
K80	NVIDIA Kepler K80 with 2 GPUs, 562 MHz, 2×12 GB, autoboost off, ECC on, HSW host	1 GPU	PGI 16.1 -ta=tesla:cc35
K40	NVIDIA Kepler K40 GPU, 745 MHz, 12 GB, autoboost N/A, ECC on, SNB host	1 GPU	PGI 16.1 -ta=tesla:cc35

Table 2. Floating-point performance and memory bandwidth (BW) of the hardware architectures under investigation

Machine	Peak GFlop/s	Peak BW [GB/s]	STREAM BW [GB/s]	VTune BW [GB/s]
BDW	422.40	76.80	60.71	68.00
HSW	240.00	68.00	55.76	61.00
SNB	128.00	51.20	35.88	43.00
IVB	128.00	51.20	40.43	43.00
P100	4759.55	720.00	550.35	N/A
K80	935.17	240.00	149.70	N/A
K40	1430.40	288.00	191.20	N/A

why we base our CPU performance portability investigations on these values. All results can be found in Table 2.

5.2 Arithmetic Intensity

To evaluate which performance boundary is hit by our JuROr code, we take a look at its arithmetic intensity in Flop per Byte [Flop/B]. Since the concept of arithmetic intensity does only make sense for individual kernels, we focus on JuROr's hotspot – the Jacobian stencil. While it takes up to 80% of the runtime in serial execution, its parallelized version still takes up to 50% of the runtime on a K40 for a 2D test case with $N_x = N_y = 4096$ grid cells in each direction. Thus, again it describes the hotspot and we can apply (8) to compute its sustainable performance with respect to its performance limiters:

$$\begin{aligned}&\text{sustainable performance } [GFlop/s]\\ &\qquad = \min(\text{sustainable BW } [GB/s] \cdot \text{A.I. } [Flop/B],\\ &\qquad\qquad \text{peak Flop/s performance } [GFlop/s])\,. \end{aligned} \tag{8}$$

In a second step, we will use performance counters to measure the achievable performance and compute its percentage share from sustainable peak:

$$\text{performance share}\,[\%] = \frac{\text{measured performance of hotspot } [GFlop/s]}{\text{sustainable performance of hotspot } [GFlop/s]}. \tag{9}$$

For determining the arithmetic intensity of the Jacobian stencil kernel, we differentiate between theoretical arithmetic intensity and measured arithmetic intensity. Here, *theoretical* arithmetic intensity refers to the traditional approach of investigating the kernel's source code and manually counting (double) floating-point operations and transferred words. While this approach works well for small regular kernels, it is very challenging for real-world codes that also employ special built-in function calls or complex data access patterns. For example, a call of the `pow` or `sin` function does not deliver an intuitive Flop per Byte ratio and, thus, is little predictable. Therefore, we also examine a *measured* arithmetic intensity of the JuROr's hotspot which is based on performance counters.

Theoretical Arithmetic Intensity. Besides counting floating-point operations, we only take the slowest data path into account, i.e., access to main memory (CPU) or global memory (GPU). For that, we evaluate the cache reuse with layer conditions to exclude corresponding data accesses. Furthermore, we verify that non-temporal stores are used on the CPU systems. Overall, for JuROr's hotspot we have:

$$\text{A.I.} = \frac{\text{floating-point operations}}{\text{data movement}} = \frac{12\ \text{Flops}}{(2\ \text{reads} + 1\ \text{write}) \cdot 8\ \text{Bytes}} = 0.500\ \frac{Flop}{B}.$$

Measured Arithmetic Intensity. The approach of measured arithmetic intensity has the advantage of being applicable for any kind of code. However, it might not reflect the best possible arithmetic intensity, since it also tracks unnecessary data transfers or occurring macho-Flop/s. To get the measured arithmetic intensity, we run the code with performance counters for double-precision floating-point operations and the transferred bytes. Since no common performance counter interface is available across the selected machines, we manually track the counters using different tools: NVIDIA's nvprof 7.5 on the NVIDIA GPU systems and Intel's VTune Amplifier 2016/2017 on the Intel CPU systems. It must be noted that a direct mapping from memory access counter values to our hotspot function is not possible since they are based on uncore events. Therefore, we use VTune's filter capabilities to track our hotspot function within the timeline view and read values from that timeline. To ease our calculations, we directly use VTune's calculated bandwidth numbers. A summary of the applied setups can be found in Tables 3 and 4.

Due to known hardware restrictions on the Intel Haswell machine [16], we are not able to use Flop performance counters on this architecture. Nevertheless, we are able to run parts of the code with the Intel Advisor tool that shall be able to measure arithmetic intensities for roofline models automatically. From the intermediate result (before crashing), we take the achieved GFlop/s number on the Haswell system. Unfortunately, the Intel Advisor is not capable of running our real-world code successfully on all architectures due to crashes. Thus, we rely on our own performance counter measurements as described above for the other architectures.

Given the counters in Tables 3 and 4, we can compute the measured arithmetic intensity as follows:

$$\begin{aligned}\text{A.I.}_{\text{CPU}} &= \frac{\text{X87} + \text{SCALAR} + \text{SSE\_PACKED} \cdot 2 + \text{256\_PACKED} \cdot 4}{(\text{RD} + \text{WR}) \cdot 64\ \text{Bytes}} \\ &= \frac{\text{X87} + \text{SCALAR} + \text{SSE\_PACKED} \cdot 2 + \text{256\_PACKED} \cdot 4}{\text{BW} \cdot \text{runtime}_{\text{hotspot}}}\end{aligned}$$

as well as

$$\text{A.I.}_{\text{GPU}} = \frac{\text{flop\_count\_dp}}{(\text{read} + \text{write}) \cdot 32\ [\text{threads per warp}]},$$

Table 3. Performance counters: Flops counters

Machine	Flops counter	Tool
BDW	FP_ARITH_INST_RETIRED.SCALAR_DOUBLE, FP_ARITH_INST_RETIRED.128B_PACKED_DOUBLE, FP_ARITH_INST_RETIRED.256B_PACKED_DOUBLE, INST_RETIRED.X87	VTune
HSW	N/A	N/A
SNB	FP_COMP_OPS_EXE.SSE_SCALAR_DOUBLE, FP_COMP_OPS_EXE.SSE_PACKED_DOUBLE, SIMD_FP_256.PACKED_DOUBLE, FP_COMP_OPS_EXE.X87	VTune
IVB	FP_COMP_OPS_EXE.SSE_SCALAR_DOUBLE, FP_COMP_OPS_EXE.SSE_PACKED_DOUBLE, SIMD_FP_256.PACKED_DOUBLE, FP_COMP_OPS_EXE.X87	VTune
P100	flop_count_dp	nvprof
K80	flop_count_dp	nvprof
K40	flop_count_dp	nvprof

Table 4. Performance counters: Bytes counters

Machine	Bytes counter	Tool
BDW	UNC_M_CAS_COUNT:RD, UNC_M_CAS_COUNT:WR	VTune
HSW	UNC_M_CAS_COUNT:RD, UNC_M_CAS_COUNT:WR	VTune
SNB	UNC_M_CAS_COUNT:RD, UNC_M_CAS_COUNT:WR	VTune
IVB	UNC_M_CAS_COUNT:RD, UNC_M_CAS_COUNT:WR	VTune
P100	dram_read_transactions, dram_write_transactions	nvprof
K80	dram_read_transactions, dram_write_transactions	nvprof
K40	dram_read_transactions, dram_write_transactions	nvprof

where

$$\text{read} + \text{write} = \text{dram\_read\_transactions} + \text{dram\_write\_transactions}.$$

Following those two approaches – of theoretical vs. measured arithmetic intensity – we present our results in the following section.

6 Results

Following the methodology introduced in Sect. 5, we present performance portability results with respect to the theoretical and measured arithmetic intensity.

6.1 Measurement Setup

In addition to the hardware setups given in Table 1, we compile all code versions with `-fast -O3`. We run all performance and counter measurements three times and take the corresponding average value while runtime deviations are below 0.6%. Furthermore, all measurements are executed on machines with exclusive access. For OpenACC runs on our CPU systems, we also enable thread binding to ensure good data affinity: `ACC_NUM_CORES=<#cores> ACC_BIND=yes MP_BIND=yes MP_BLIST=0,1,<...#cores-1>`.

Since selecting OpenACC loop schedules is left to the compiler, Table 5 gives an overview on the PGI compiler's choice for the Jacobian stencil on different hardware setups. For our CPUs, the outer loop of the Jacobian loop nest gets distributed across `gangs` (i.e. CPU cores), while the compiler attempts to vectorize the inner loop. Contrarily, the compiler choses a two-dimensional work distribution on the GPUs: Each dimension gets distributed across the GPU's multiprocessors (`gangs`) and the double-precision logic units (`vector`). While the overall thread tile size is the same across all GPUs, i.e., 128 threads per block, the compiler selects different distributions within the tiles for Kepler and Pascal GPUs.

Table 5. Loop schedules for loop nests of Jacobian stencil kernel chosen and reported by the PGI compiler

Machine	Outer loop	Inner loop
BDW	gang	vector sse + prefetching
HSW	gang	vector sse + prefetching
SNB	gang	vector sse + prefetching
IVB	gang	vector sse + prefetching
P100	gang vector(32)	gang vector(4)
K80	gang vector(4)	gang vector(32)
K40	gang vector(4)	gang vector(32)

6.2 Theoretical and Measured Arithmetic Intensity

Results for the theoretical and measured arithmetic intensity of the Jacobian stencil are presented in Table 6. Values of the measured arithmetic intensity show only little deviation with values in the range of 0.332 to 0.498 Flop/B across all architectures. In addition, they are roughly in line with the theoretical arithmetic intensities of 0.500 since the Jacobian stencil does not exhibit any special built-in functions or macho-Flop/s.

Table 6. Theoretical and measured A.I. of the Jacobian stencil kernel

Machine	Theoretical A.I. $[\frac{Flop}{B}]$	Measured A.I. $[\frac{Flop}{B}]$	Performance limiter
BDW	0.500	0.340	Memory bandwidth
HSW	0.500	0.332	Memory bandwidth
SNB	0.500	0.386	Memory bandwidth
IVB	0.500	0.354	Memory bandwidth
P100	0.500	0.498	Memory bandwidth
K80	0.500	0.416	Memory bandwidth
K40	0.500	0.418	Memory bandwidth

6.3 Performance Portability

As an overview, two exemplary roofline models for JuROr running on the Broadwell CPU in Fig. 2 and the Pascal GPU in Fig. 3 illustrate the theoretical intensity (vertically dashed line) and measured arithmetic intensity (circle marker) while also visualizing the performance limiters as rooflines. This representation also shows the achieved performance (circle marker) in comparison to the sustainable memory bandwidth.

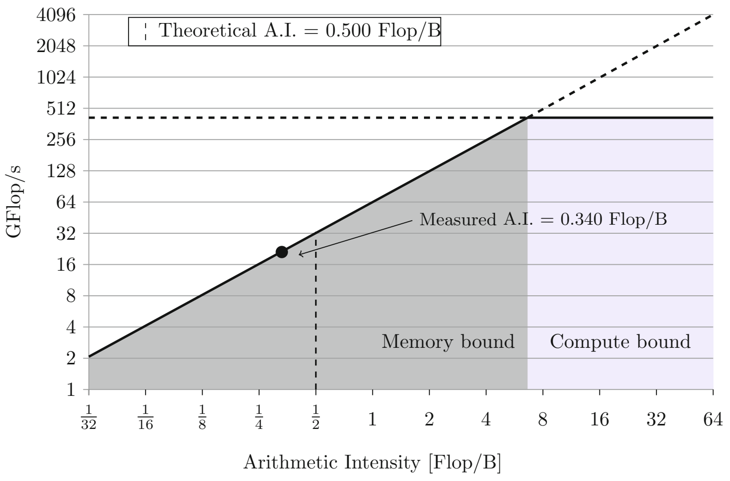


Fig. 2. Roofline of BDW based on data set size of $N_x = N_y = 4096$

For our detailed analysis, we list the absolute performance numbers in Table 7 that are derived by our performance counter measurements running the JuROr code. All these numbers, i.e., GFlop/s, GB/s and runtime in seconds, highly differ across the architectures giving the impression of having non-portable code with respect to performance.

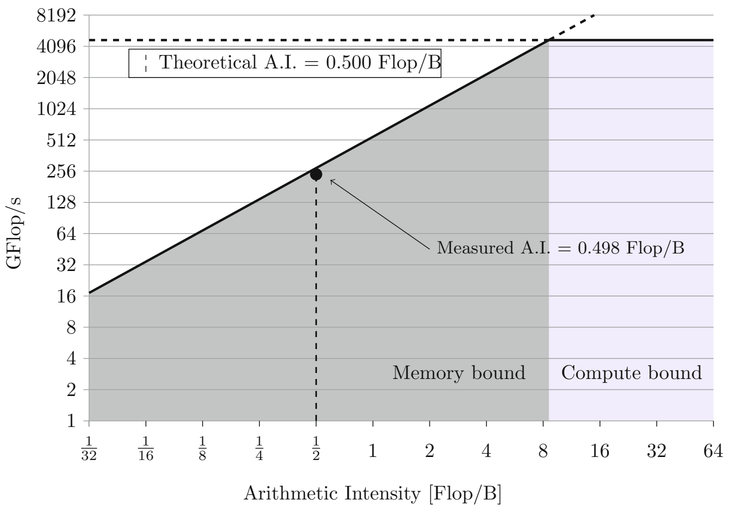


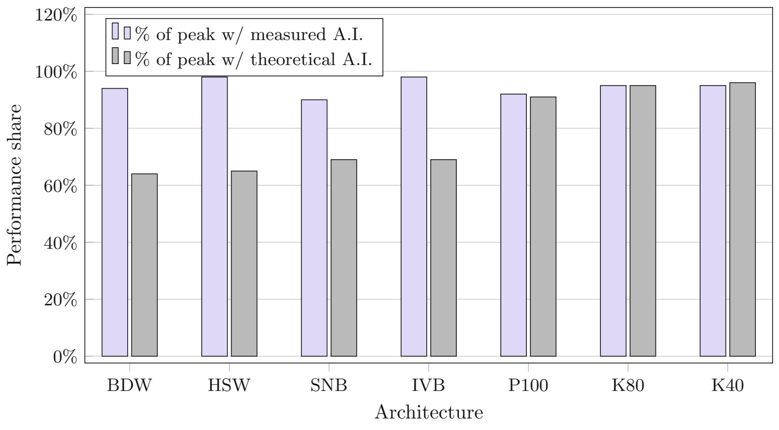
Fig. 3. Roofline of P100 based on data set size of $N_x = N_y = 4096$

Table 7. Flop/s, memory bandwidth and runtime measurement for Jacobian stencil kernel. Bandwidths given in brackets are based on ECC overhead.

Machine	Measured GFlop/s	Measured BW [GB/s]	Kernel runtime [s]
BDW	21.66	63.71	4.97
HSW	19.81	59.59	5.29
SNB	14.93	38.65	8.54
IVB	14.90	42.04	7.70
P100	251.77	505.14	0.47
K80	71.17	170.91 (−29.08)	1.65
K40	91.47	218.79 (−36.30)	1.29

However, in the following, we express performance portability as performance share to sustainable peak by applying our definition in (9). These results are illustrated in Fig. 4.

Looking at the theoretical arithmetic intensities, the Jacobian stencil achieves 64% to 69% of sustainable memory bandwidth (given by Intel VTune's micro benchmarks) across the CPUs. For the GPU systems, it achieves higher performance shares that range from 91% to 96% with respect to the GPU-STREAM results. Since the measured arithmetic intensities are slightly below the theoretical values, they also assume a lower sustainable peak performance in GFlop/s (exemplary illustrated in Fig. 2). Therefore, we see higher performance shares for the measured arithmetic intensities ranging from 90% to 98% on the CPUs with respect to Intel VTune's bandwidth micro benchmark and from 104% to 108% with respect to the OpenMP STREAM benchmark results. Thus, our



	BDW	HSW	SNB	IVB	P100	K80	K40
GFlop/s w/ theor. A.I.	34.00	30.50	21.50	21.50	275.17	74.85	95.60
GFlop/s w/ meas. A.I.	23.12	20.27	16.60	15.24	274.30	62.34	79.93
Measured GFlop/s	21.66	19.81	14.93	14.90	251.77	71.17	91.47

Fig. 4. Performance share of all considered architectures for $N_x = N_y = 4096$

hotspot delivers higher bandwidth measurements than the STREAM benchmark which may be due to additional transferred bytes for prefetching. For the GPU performance shares, initially, we see a similar behavior with values from 92% to 114%. When investigating the appearance of the GPU performance shares above 100% further, i.e., for the two Kepler architectures K80 and K40, we find that NVIDIA's device memory performance counters also track transactions caused by ECC overhead (cf. Table 7). Since these extra ECC bytes do not contribute to the bandwidth achieved by the application, we subtract corresponding values (counters `ecc_transactions/ ecc_throughput`) from the measured bandwidth of the Jacobian stencil. In contrast, the Pascal architecture supports ECC natively and, hence, does not show ECC effects on bandwidth. With that, we get more realistic performance shares for JuROr of 92% to 95% across the GPUs.

Overall, although absolute performance numbers suggest otherwise, the results that are based on the specific hardware and software characteristics show that for our real-world OpenACC code the PGI compiler is capable in producing performance portable code across different target architectures with a single source code base.

7 Conclusion and Outlook

In the context of the OpenACC-parallel real-world C++-code JuROr that simulates smoke propagation based on computational fluid dynamics, we investigated the performance portability of its memory-bound hotspot using PGI's OpenACC across four Intel CPUs and three NVIDIA GPUs.

For our analysis of performance portability, we setup roofline models for all architectures and computed the arithmetic intensity of the code's hotspot – the Jacobian stencil. We examined this theoretical arithmetic intensity, as well as measured arithmetic intensities that were obtained using performance counters for floating-point operations and memory transfers. Our measured arithmetic intensities are in the range of 0.332 to 0.498 Flop/B for all architectures and, thereby, roughly in line with the theoretical arithmetic intensity of 0.500 Flop/B.

Using the theoretical arithmetic intensity, we obtained 64% to 69% of sustainable bandwidth on the CPUs and 91% to 96% on the GPUs. Regarding the measured arithmetic intensities, the performance shares increased to 90% to 98% on the CPUs and remained roughly constant with 92% to 95% on the GPUs referring to the according STREAM bandwidths, respectively. Our investigations show that it is important to account for ECC overhead in memory bandwidth on Kepler GPUs when using NVIDIA's device memory performance counters. Pascal GPUs lift this problem by natively supporting ECC in hardware.

Due to the similar performance shares across architectures, our OpenACC parallelization of JuROr shows good performance portability relying on the PGI compiler. While hand-tuned or low-level code might generally achieve higher performance, our OpenACC approach gives us the possibility to maintain one source code base for different architectures while still delivering good performance.

In future, to achieve further parallelization and acceleration, we will constantly optimize the code for both, CPU and GPU usage and model the data transfer for the roofline. Moreover, we will investigate OpenACC performance on AMD GPUs. While we could already show that our OpenACC code is runnable on AMD Tahiti GPUs, problems with the measurement infrastructure hindered us in presenting portability results in this paper. Currently, we are working on a 3D code to handle 3D geometries, where we will further include handling of inner boundaries to expand the code to complex 3D geometries. Complex 3D geometries (e.g., several rooms) will then be used for the validation of the OpenACC code.

Acknowledgements. This study was performed within the project ORPHEUS funded by the Federal Ministry of Education and Research (BMBF) Program on 'Research for Civil Security - Protection and Rescue in complex Disaster Situations' (funding code 13N13266). Some simulations were performed with computing resources granted by RWTH Aachen University under project rwth0207.

References

1. BMBF funded research project, Optimierung der Rauchableitung und Personenführung in U-Bahnhöfen: Experimente und Simulationen (ORPHEUS) - Teilvorhaben: Brand- und Personenstromsimulationen in unterirdischen Verkehrsstationen (2015–2018). http://www.orpheus-projekt.de

2. Williams, S., Waterman, A., Patterson, D.: Roofline: an insightful visual performance model for multicore architectures. Commun. ACM **52**(4), 65–76 (2009)
3. Han, L., et al.: FireGrid: an e-infrastructure for next-generation emergency response support. J. Parallel Distrib. Comput. **70**(11), 1128–1141 (2010)
4. Koo, S.-H.: Forecasting fire development with sensor-linked simulation, Dissertation, University of Edinburgh (2010)
5. Glimberg, S.L., Erleben, K., Bennetsen, J.: Smoke simulation for fire engineering using a multigrid method on graphics hardware. In: VRIPHYS, pp. 11–20. Eurographics Association (2009)
6. Daniel, N., Rein, G.: The Fire Navigator: forecasting the spread of building fires on the basis of sensor data, FPE Extra Issue 3, March 2016. http://www.sfpe.org/general/custom.asp?page=FPEExtraIssue3
7. Pennycook, S.J., Hammond, S.D., Wright, S.A., Herdman, J.A., Miller, I., Jarvis, S.A.: An investigation of the performance portability of OpenCL. J. Parallel Distrib. Comput. **73**(11), 1439–1450 (2013)
8. Lopez, M.G., Larrea, V.V., Joubert, W., Hernandez, O., Haidar, A., Tomov, S., Dongarra, J.: Towards achieving performance portability using directives for accelerators. In: Third Workshop on Accelerator Programming Using Directives (WACCPD), pp. 13–24 (2016)
9. Sabne, A., Sakdhnagool, P., Lee, S., Vetter, J.S.: Evaluating performance portability of OpenACC. In: Brodman, J., Tu, P. (eds.) LCPC 2014. LNCS, vol. 8967, pp. 51–66. Springer, Cham (2015). doi:10.1007/978-3-319-17473-0_4
10. Herdman, J.A., Gaudin, W.P., Perks, O., Beckingsale, D.A., Mallinson, A.C., Jarvis, S.A.: Achieving portability and performance through OpenACC. In: First Workshop on Accelerator Programming using Directives, pp. 19–26. IEEE Press (2014)
11. Nicolini, M., Miller, J., Wienke, S., Schlottke-Lakemper, M., Meinke, M., Müller, M.S.: Software cost analysis of GPU-accelerated aeroacoustics simulations in C++ with OpenACC. In: Taufer, M., Mohr, B., Kunkel, J.M. (eds.) ISC High Performance 2016. LNCS, vol. 9945, pp. 524–543. Springer, Cham (2016). doi:10.1007/978-3-319-46079-6_36
12. Calore, E., Gabbana, A., Kraus, J., Schifano, S.F., Tripiccione, R.: Performance and portability of accelerated lattice Boltzmann applications with OpenACC. Concurr. Comput. Pract. Exper. **28**(12), 3485–3502 (2016)
13. Wang, Y., Qin, Q., See, S.C.W., Lin, J.: Performance portability evaluation for OpenACC on Intel Knights Corner and Nvidia Kepler. In: HPC China (2013)
14. Chorin, A.: Numerical solution of the Navier-Stokes equations. Math. Comput. **22**, 745–762 (1968)
15. Smagorinsky, J.: General circulation experiments with the primitive equations. Mon. Weather Rev. **91**(3), 99–164 (1963)
16. JURECA, Jülich Research on Exascale Cluster Architectures. http://www.fz-juelich.de/ias/jsc/EN/Expertise/Supercomputers/JURECA/JURECA_node.html
17. Top500.org, Top500 List, November 2016. https://www.top500.org/list/2016/11/
18. McCalpin, J.D.: Memory bandwidth and machine balance in current high performance computers. IEEE Comput. Soc. Techn. Committee Comput. Archit. (TCCA) Newsl. 19–25 (1995). https://www.cs.virginia.edu/stream/
19. Deakin, T., McIntosh-Smith, S.: GPU-STREAM v1.0/ v3.1. https://github.com/UoB-HPC/GPU-STREAM

20. Deakin, T., Price, J., Martineau, M., McIntosh-Smith, S.: GPU-STREAM v2.0: benchmarking the achievable memory bandwidth of many-core processors across diverse parallel programming models. In: Taufer, M., Mohr, B., Kunkel, J.M. (eds.) ISC High Performance 2016. LNCS, vol. 9945, pp. 489–507. Springer, Cham (2016). doi:10.1007/978-3-319-46079-6_34
21. Danalis, A., Marin, G., McCurdy, C., Meredith, J., Roth, P., Spafford, K., Tipparaju, V., Vetter, J.: The scalable heterogeneous computing (SHOC) benchmark suite. In: Proceedings of the Third Workshop on General-Purpose Computation on Graphics Processors (GPGPU 2010), pp. 63–74 (2010)