GPU-Accelerated Particle-in-Cell Code on Minsky

Andreas Herten(✉), Dirk Brömmel, and Dirk Pleiter

Forschungszentrum Jülich, JSC, 52425 Jülich, Germany
{a.herten,d.broemmel,d.pleiter}@fz-juelich.de

Abstract. Particle-in-cell (PIC) methods are widely used on today's supercomputers. In this paper we consider JuSPIC, an application for which good scaling properties could be demonstrated on a 6PFlop/s BlueGene/Q system. We report on efforts to port this application to emerging supercomputing architectures based on IBM POWER processors and NVIDIA graphics processing units.

Keywords: POWER8 · GPU acceleration · Performance analysis · Minsky · OpenPOWER · NVIDIA Tesla P100

1 Introduction

Numerical methods are key for investigating laser-plasma interactions due to the non-linear nature of the problem and the non-trivial geometries implied in the problem. Most commonly used is the particle-in-cell (PIC) method for simulating the motion of charged and neutral plasma particles. PIC codes solve Maxwell's equations on a grid using currents and charge densities calculated by weighting discrete particles onto the grid. In each update step, position and momentum of each particle are updated based on forces acting on them, which are obtained from self-consistently calculated fields. The development and use of this methods goes back into the 1950-60s [13].

Due to its intrinsic high level of parallelism, PIC applications simulating millions of particles are good candidates for massively-parallel computer architectures. In this contribution we focus on a special type of such architectures, which are based on IBM POWER processors and graphics processing units (GPU) from NVIDIA. Such solutions have become available only recently and the ecosystem for these is still emerging. In this paper we will explore the performance for JuSPIC, a PIC code developed at Jülich Supercomputing Centre, on IBM S822LC servers (also known as "Minsky"), which features novel tight integration of processor and GPU based on the new NVLink technology.
This paper makes the following contributions:

1. A port of JuSPIC to the Minsky platform and report of experiences for different porting strategies.

J.M. Kunkel et al. (Eds.): ISC High Performance Workshops 2017, LNCS 10524, pp. 205–219, 2017.
https://doi.org/10.1007/978-3-319-67630-2_17

2. Analysis of the performance as a function of different hardware settings employing a semi-empirical performance modelling approach.
3. Report on experience in optimisation for this new platform.

The paper is organised as follows: After the introductory Sect. 2 we introduce the architecture of the compute platform used in this paper in Sect. 3. We report in Sect. 4 on porting JuSPIC to GPUs, before presenting performance results in Sect. 5. After providing an overview on related work in Sect. 6 we summarise our results and present our conclusions in Sect. 7.

2 JuSPIC

JuSPIC [1,8], the Jülich Scalable Particle-in-Cell (PIC) code is used to simulate particles in electromagnetic fields. Like other PIC codes, it can be used as a numerical tool in the field of intense laser-plasma interaction, e.g. to simulate the generation of energetic electrons and ions with help of the radiation field of a laser to study approaches for table-top particle accelerators. The code is based on H. Ruhl's Plasma Simulation Code (PSC) [7] and further developed at the Jülich Supercomputing Centre (JSC) mainly for testing new HPC architectures and programming models. But it has also been used to support experimental investigations of relativistic, highly non-linear laser-plasma interaction acting as Terahertz light-source [12].

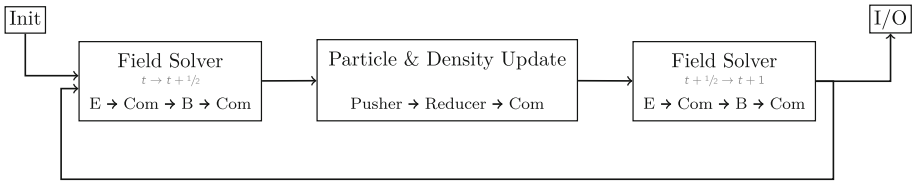

Fig. 1. Core steps of JuSPIC. After initializing (*Init*), the E and B fields are computed and communicated for a half-time step (*Field Solver*). Following this, the particle and density information is updated. The *Pusher* uses information from the grid to update particle information, the *Reducer* again takes this information to update the grid. A communication step ensues, before this iterations concludes with another invocation of the Field Solver. The algorithm is started again at the first Field Solver. Data can be written/output after one chain (*I/O*).

The interaction between fields and plasma is described by the relativistic Vlasov equation and Maxwell's equations. JuSPIC uses a regular mesh for the Maxwell fields and particle charge and current densities that are then integrated using the Finite-Difference-Time-Domain (FDTD) scheme. Plasma particles are modelled via distribution functions of quasi-particles with continuous coordinates within the mesh. Finite element approximations of the distribution functions are then used to integrate the Vlasov equation along the particle trajectories [6,15].

Omitting details about PIC codes in general, let us note that due to the required high particle numbers, the coupled system of Vlasov and Maxwell equations requires two very time-consuming steps: First, the particle update, computing new particle positions and velocities that requires the interpolation of the electromagnetic fields from their mesh coordinates to the position of the quasi-particles. We denote this part the *pusher*; it is the main focus of our GPU acceleration. The second step is the reduction of the continuous charge and particle densities of each quasi-particle onto the mesh (the *reducer*). This sparse reduction poses a special challenge since it happens irregularly on very large fields and is a non-local operation.

JuSPIC is written in modern Fortran and parallelised with MPI and OpenMP. The basic operation of the algorithm is outlined in Fig. 1. It is capable of scaling to the full JUQUEEN system, a 28-rack IBM BlueGene/Q using 1.8M hardware threads and listed in the High-Q Club [2]. JuSPIC is Open Source [3].

3 Compute Platform

The performance results shown in this paper have been obtained in large parts on **JURON**, a cluster based on IBM S822LC servers. Each server comprises 2 IBM POWER8 processors with 2 NVIDIA P100 GPUs each. One processor and 2 GPUs are interconnected in a ring topology using NVLink.

In this server, 4 out of 8 DMI channels per processor are used for attaching Centaur memory chips to which the DDR4 memory is attached. The GPUs use a new high-bandwidth memory technology called HBM2, which allows for significantly higher memory bandwidth but limited memory capacity. Due to the new NVLink connections, the bi-section bandwidth for data transport between processor and GPUs is similar to the processor-memory bandwidth.

Key hardware performance numbers are summarised in Table 1.
Additionally, two x86 systems are referenced in parts of this paper:

JUHYDRA. A testing system with 2 Intel Xeon E5-2650 CPUs (2 GHz) and 2 NVIDIA Tesla K20Xm and K40m GPUs, each, attached via PCIe.

JURECA. Jülich's large multi-purpose supercomputer with nodes with 2 Intel Xeon E5-2680 CPUs (2.5 GHz) and 2 PCIe-attached NVIDIA Tesla K80 GPUs (appearing as 2 GPU devices, each).

4 Acceleration for GPUs

For JuSPIC, a hybrid approach in acceleration for GPUs has been chosen. Parts of the code have been ported employing the OpenACC programming model, parts use CUDA Fortran.

Table 1. Selected (nominal) hardware parameters for the IBM S822LC servers.

IBM POWER8 processor	
Number of processors	2
Default clock frequency	3491 MHz
Total number of cores	20
Aggregate throughput of floating-point operations (DP)	559 GFlop/s
Aggregate memory read bandwidth	77 GByte/s
Aggregate memory write bandwidth	154 GByte/s
Aggregate memory capacity	256 GByte
NVIDIA P100 GPU	
Number of GPUs	4
Default clock frequency	1328 MHz
Total number of SMs	224
Aggregate throughput of floating-point operations (DP)	19038 GFlop/s
Aggregate memory bandwidth	2880 GByte/s
Aggregate memory capacity	128 GByte
Aggregate CPU-GPU bandwidth	160 GByte/s

4.1 OpenACC

In OpenACC, code to be accelerated is annotated with statements which are interpreted by a capable compiler to create programs for GPUs or other many-core systems. Since these compiler directives are usually in the form of comments in the code, portability across many different systems is achieved. Depending on the capabilities of the compiler, different accelerator architectures can be targeted from the same code base. For JuSPIC, the PGI compiler (version 16.10) is used. It features both support for OpenACC and CUDA Fortran.

OpenACC is used in JuSPIC to move data between the host and the device. Data regions are created to move data, partly asynchronously, to device or host memory depending on where the next part of the program operates. As an example, the first region is created by `!$acc enter data async copyin(e,b,ji)` which creates a copy of the matrices `e`, `b`, and `ji` on the GPU asynchronously. Data is transferred back, and also updated as-needed in subsequent steps of the algorithm. In the absence of any acceleration device, the compiler produces a version of the program in which the data regions are omitted.

OpenACC is used furthermore to port the time propagation of the electric and magnetic fields to the GPU. For each field, a three-fold nested loop updates three directions in space for different indexes. The `kernels` directive is used, giving the compiler the most freedom to accelerate the scope of the multiplication: `!$acc kernels loop collapse(3) present(e,b)`. The three loops are merged into a loop of one level by the `collapse` clause; the electro-magnetic

fields are already on the device, since they have been copied asynchronously beforehand.

4.2 CUDA Fortran

By far the most compute-intensive part of JuSPIC is the update of particle velocities and momenta in the pusher. In an initial attempt, this part was also enabled for the GPU with OpenACC. Unfortunately, the structure of the code was yet to be supported by the compiler. Many structured data types (e.g. `particles(i)%x(0)`), operations on whole fields at once (`b=a*2`, where `a` and `b` are both fields), as well as the amount of operations on the data prevented efficient code generation by the compiler. The algorithm needed to be adapted vastly to make it more *recognisable* for the compiler. With those changes, the initially non-compiling OpenACC pusher did compile, but ran very slow. Only a version of the algorithm, which had each operation on a whole field replaced by operations on the individual elements of the field ran reasonably well. We decided to rather return to the original pusher algorithm and use CUDA Fortran to port it to the GPU. [14]

CUDA Fortran is a Fortran interface to NVIDIA's CUDA C/C++ programming model. It is developed by PGI and available in their Fortran compilers. It is modelled closely alongside CUDA C/C++ and additionally implements features of the Fortran programming language, like operations on whole fields.

Using CUDA Fortran, the pusher kernel is ported to the GPU. The original, serial code is taken, the `do` loop over the particles removed, and replaced with `threadIdx %x`-based indexing in typical CUDA . Compatibility to systems without GPUs or CUDA Fortran is ensured by wrapping the specialised GPU parts with pre-processor macros. This way, either the original pusher loop is called in the absence of CUDA Fortran, or the GPU kernel is called with `call gpupusher<<<dim3(nBlocks, 1, 1), dim3(nThreads, 1, 1)>>>`

Writing the pusher kernel in CUDA Fortran enables the evaluation of different strategies of handling the data of the particles. We study four cases:

1. All particles are stored in a single field, one particle after another, the field is copied to and from the GPU with CUDA Fortran (*Initial*).
2. As above, but data is copied using OpenACC `copy` statements (*Experiment One (Exp 1)*).
3. As above, data is copied with OpenACC statements, from pinned (zero-copy) host memory (*Experiment Two (Exp 2)*).
4. Instead of one field holding all data from all particles, spatial and momentum components for all particles are stored in separate fields (six fields in total); data is copied with CUDA Fortran statements (*Structure-of-Array Approach (SoA)*).

The results of the four cases are summarised in Table 2, averaged per invocation of the GPU pusher. Shown is JURON (with a P100 GPU) and, as a comparison, JUHYDRA (using a K40 GPU). In the table, *Kernel* denotes the

Table 2. Runtimes of the particle pusher of JuSPIC on the GPU for different methods of data handling. **Legend:** *Allocate* – Allocate host-side memory region; *LL2F* – Convert linked list data structure to field; *H2D* – Transfer data from host to device; *D2H* – Transfer data from device to host; *F2LL* – Copy data from field to linked list data structure.

in µs	$\sum$	Allocate	LL2F	H2D	Kernel	D2H	Others	F2LL
				JURON				
Initial	8039.71	–	567.27	81.86	83.71	61.70	350.19	6884.87
Exp 1	10434.51	–	353.08	80.39	81.82	91.48	379.87	9440.10
Exp 2	9695.39	563.50	526.69	79.19	82.57	72.38	107.87	7972.61
SoA	7810.95	0.94	843.69	65.74	76.57	53.03	376.25	6386.24
				JUHYDRA				
Initial	4955.59	–	907.63	267.11	229.27	207.62	735.92	2600.14
Exp 1	4687.25	–	763.99	231.58	228.51	197.59	804.41	2455.00
Exp 2	5328.37	576.97	1026.94	223.56	229.65	192.17	23.17	2651.33
SoA	4879.61	1.05	785.84	204.14	207.58	173.40	826.55	2673.97

runtime of the GPU pusher kernel itself. *H2D* and *D2H* shows the time spent for copying data to and from the GPU, respectively. *Others* incorporates the time the GPU is not processing or copying data – the device usually waits for instructions or synchronises. *Allocate* refers to the time needed to allocate a memory region on the host side (for pinned memory in case of *Exp 2* and for the individual SoA fields in case of *SoA*). *LL2F* and *F2LL* are pre-processing steps: The format in which JuSPIC stores particles is a linked list, with each particle including a pointer to the next. To enable coalesced loads on the GPU, data is copied from a linked list to a field before GPU kernel invocation and from a field back to a linked list after completion – *LL2F* and *F2LL*, respectively. The relatively high runtimes of these parts are analysed and discussed in Sect. 4.3.

Looking at each of the architectures, the runtimes of data copies and kernels in the SoA approach is in all cases the fastest. The data layout is not only beneficial for efficient execution but also for data movements. In case of preparing the data in the LL2F step, SoA is the slowest. The explicit filling of two three-vectors (position, momentum) to individual and distinct memory locations from one packed source particle seems to take more time then the simple copy from one memory position to another. For the post-processing F2LL step, the case appears to be inverted (see also Sect. 4.3). For Exp 2, where pinned memory is used, the overhead in form of waiting for data is the smallest. In this case, the CUDA runtime can omit safety measures and directly access the data. Unfortunately, the benefit in time is diminished by the overhead of allocating the pinned memory.

Comparing the two GPU (and also system) architectures shows that the P100 has, in nearly all cases, the lesser runtime (with F2LL being an important

deviation, see Sect. 4.3). About a factor of 3 in performance gain can be obtained compared to using a K40m GPU.

Further potential optimization can be integrated combining the distinct benefits of the individual approaches. In the current implementation of Exp 2, pinned memory is allocated once before the pusher loop and deallocated afterwards. Moving the data region one level up would enable a more data-economical approach: Pinned memory can be allocated once at the beginning of the algorithm and only reallocated if the number of particles changes during the run of the algorithm. This is a strategy already employed by the SoA version. In the SoA version, though, pinned memory has not yet been tested. Since pinned memory leads to less overhead during the GPU pusher runtime, this could be a very rich modification.

In the current state of the algorithm, data transfers to and from the device take a significant fraction of time on the GPU. Our hope is that once the next part of the algorithm, the *reducer*, is ported to the GPU as well, most of the data transfers can be saved, since the majority of the data would stay on the GPU for different iterations. This is the case as well for the changes in data layout (linked list, fields). The Unified Memory feature of CUDA, which uses efficient page faults in CUDA 8.0 and on Pascal GPUs, promises to be a productive technique to reduce data migrations to a minimum.

4.3 Investigation of Slow Data Layout Conversion

Striking in the numbers of Table 2 are the times taken for copying data from fields to linked lists, F2LL. The runtimes of 6ms to 9ms for each invocation of the pusher are about 2.5× higher than on a x86 system.

In a benchmark study, we investigate the reason for this. We create linked lists for different numbers of particles, fill them, and destroy them again. We study different architectures: the two Intel-based architectures of JURECA and JUHYDRA and the POWER8NVL system of JURON. Additionally, different compilers are tested: the PGI Fortran compiler with and without an MPI wrapper[1]; the Fortran compiler from the GNU Compiler Collection (`gfortran`, *GCC*) with and without MPI; and the XL Fortran compiler, *XLF*, if available. The part of the benchmark code which fills the linked list – `add_one_to_list` – is implemented the following way:

```
allocate(list%tail%next)
nullify(list%tail%next%next)
list%tail%next%particle = particle
list%tail => list%tail%next
```

Using this scheme, each particle is added to the list iteratively.

Figure 2 displays the time spent for adding one particle to a linked list for different total list sizes and different compiler and system configurations. Systematically, on each system, GCC-compiled benchmarks take the least time for

[1] The compiler ships with its own compiled OpenMPI version.

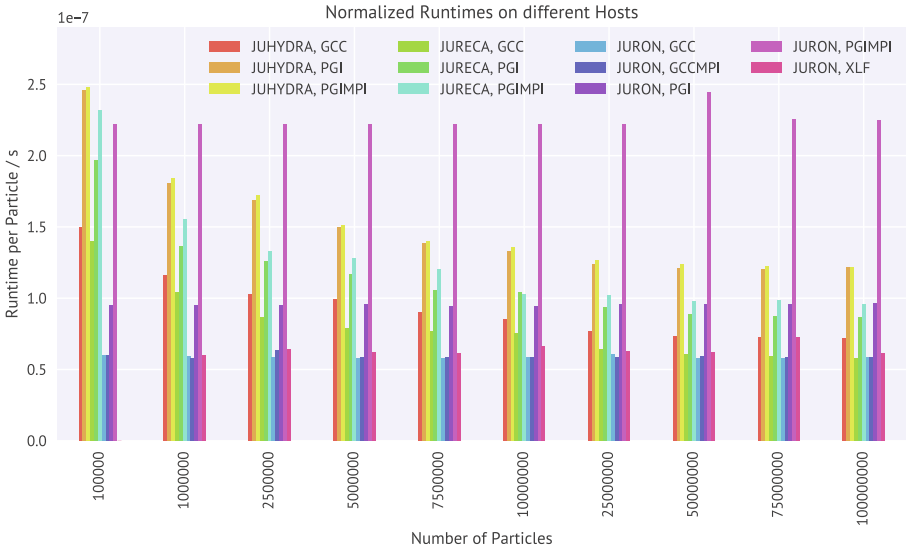

Fig. 2. Runtimes for adding a particle to a linked list vs. total number of added particles to the list for different compilers. Note: The data point of XLF for 100 000 particles is missing as the program reported no meaningful time in this case.

the operation. The PGI case is always 30% to 60% slower, compared to the respective runtime of a GCC-compiled program. On JURON, the XLF compiler produces code which is about as fast as the GCC version. The time spent for adding one particle to the list decreases with growing number of total particles for most of the versions tested on x86 – in these cases, there seems to be a undetermined overhead present which becomes more and more negligible[2].

The most important feature of the plot is the runtime of benchmarks compiled with MPI and underlying PGI compiler on JURON. The *PGIMPI* case is consistently at least 2× slower then the comparable PGI version. This is remarkable, since in both versions the same `pgfortran` compiler is used with identical compiler flags and resulting object code – just with an additional MPI wrapper in one case.

Using PAPI [17,18], we measure the number of instructions completed in the case of destroying a list in the benchmark. We choose destruction of a list over creation because this part of the code involves even fewer overhead. The performance counter measured with PAPI is `PAPI_TOT_INS`, which maps on JURON to `PM_INST_CMPL` and on JUHYDRA to `INSTRUCTION_RETIRED`. Table 3 summarises the measurements for different compilers. PGIMPI* in this case is a custom OpenMPI 2.0 version which is explicitly compiled for the benchmark test case. JuSPIC has yet to be run with this custom MPI version, as it is not officially

[2] Measurements show that the number of completed instructions is linear with the number of particles, so the overhead seems to come from the timing operation.

Table 3. Number of instructions of and time spent for clearing a linked list. The values are for a list of 10 million elements, but shown normalised per particle (pP). Time per particle is rounded to the nearest integer.

System	JURON						JUHYDRA	
Compiler	GCC	GCCMPI	PGI	PGIMPI	PGIMPI*	XLF	PGI	PGIMPI
Time pP/ns	36	37	46	154	–	41	32	32
Instructions pP	121	121	243	462	243	121	210	210

supported by the PGI compiler version. The version of GCC is 5.4.0, PGI is of version 16.10 on JURON and 16.3 on JUHYDRA. On JURON, GCCMPI uses OpenMPI 2.0.2 and PGIMPI uses OpenMPI 1.10.2 – the version shipped with the PGI compiler. The XLF version is 16.1.0. The MPI version used together with PGI on JUHYDRA is OpenMPI 1.8.1.

The results of runtimes already seen in Fig. 2 correlate with the number of completed instructions. The PGI compiler produces code which is slightly slower compared to the GCC version; in terms of instructions about twice as many are completed. The PGIMPI case doubles the number of instructions of the MPI-less PGI version further. At the same time, the number of instructions per cycle is reduced from about 2 to about 0.8.

The source code for creating and destroying linked lists is very simple. The only other operation apart from changing pointers is *allocation* and *deallocation*, respectively, see the code snippet at the beginning of Sect. 4.3. We suspect that the reason for the long runtimes lies in this allocation and deallocation.

When MPI is loaded on top PGI, a number of different libraries are linked additionally compared to the bare PGI case. We reckon that one of the libraries loaded replaces the memory allocation call with a particularly slow one in the JURON case. We test this assumption by using the linker's environment variable `LD_PRELOAD` to force loading of a specific `malloc` call when invoking JuSPIC– we use `LD_PRELOAD=/lib64/libc.so.6`. This indeed removes the instruction overhead compared to the bare PGI case entirely.

A second test replaces the PGI-shipped OpenMPI version with our own custom-compiled OpenMPI, PGIMPI*. Also in this case, the overhead is reduced to zero. The overhead hence seems to be tightly connected to the specific MPI version shipped with the PGI compiler on the POWER system[3].

While the strategies employed during investigation (`LD_PRELOAD`, custom OpenMPI) can be easily applied to the benchmark case, further in-detail studies are needed for the whole of JuSPIC to judge all ramifications and side-effects.

For the time being, we consider time spent for converting between linked lists and fields in the F2LL and LL2F regions an overhead, which is anomalously high in the system/compiler configuration at hand. The initial mitigation strategy in the future will be to test our custom OpenMPI version thoroughly with JuSPIC; a bug report with the vendor of the compiler has been filed. In a mid-range time

[3] True for both PGI 16.10 and PGI 17.1.

frame we hope to retire the linked list implementation of particles in JuSPIC in favour of a field approach globally, to better match the requirements of modern many-core architectures. Linked lists were chosen in the original design of JuSPIC because they were the fastest of the tested options to perform the sparse global reduction after the particle pusher in a multi-threaded approach [8].

5 Performance Modelling

To compare different GPU architectures and understand the behaviour of JuSPIC, we study the code in the scope of a simple performance model.

5.1 Determination of Effective Bandwidth

Our model incorporates the exchanged information of the kernel for a given amount of processed particles. It is a lower limit of the achieved bandwidth of the program. The model is parameterised by

$$t(N_{\text{part}}) = \alpha + I(N_{\text{part}})/\beta \ , \tag{1}$$

where α and β are fit parameters and I is the exchanged information, resulting in a kernel runtime of t. We call β the *effective bandwidth*. The tested version of the pusher kernel reads 572 Byte and stores 40 Byte per particle.

Figure 3 shows results of the performance model in Eq. 1 for different numbers of particles, leading to different amounts of exchanged information. Four different

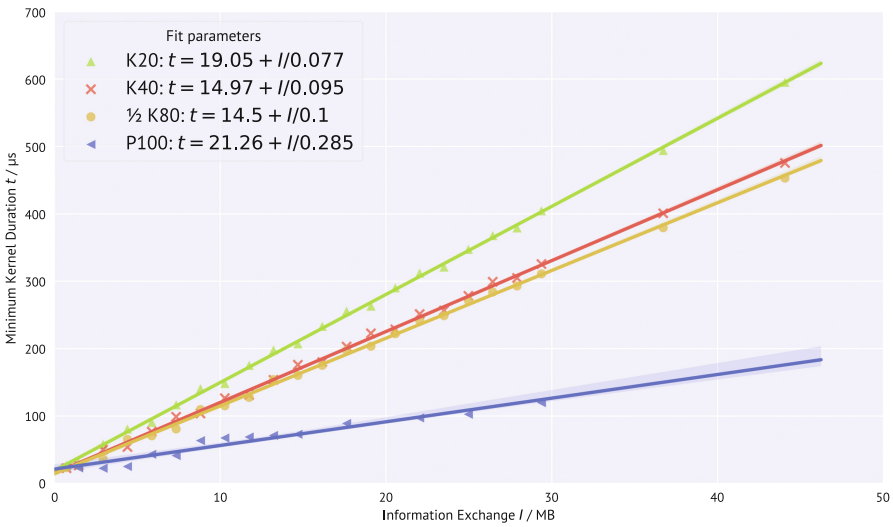

Fig. 3. GPU kernel duration as a function of exchanged information for four different GPUs: K20, K40, ½ K80, and P100. Shown in the legend are fit parameters to the performance model.

NVIDIA GPUs are studied: Tesla K20 and K40 devices (both on JUHYDRA), one part of a Tesla K80 GPU (1/2 K80; on JURECA), and the Pascal P100 on JURON. In all cases, the GPU boost feature, which automatically increases the graphics clock rate of the devices, has been disabled and the clock fixed to its default size.

From the linear regressions, which are superimposed in Fig. 3, the effective bandwidths as of the performance model can be deduced:

K20: 77 GB/s K40: 95 GB/s 1/2K80: 100 GB/s P100: 285 GB/s

The utilised bandwidth is higher for the K80 (42%) than for the K40 (33%), since the maximum available bandwidth is lower for the K80 (240 GB/s) than for the K40 (288 GB/s). The efficiency of JuSPIC on this device is higher. The same bandwidth utilization of the K80 is achieved for the P100 device: About 40% of the available 720 GB/s bandwidth is used.[4] The absolute value of utilised bandwidth is higher (285 GB/s), caused by the new Pascal architecture features: The higher bandwidth to the HBM2 memory offers more throughput of data, while the greater number of multiprocessors leads to more computations per time and more threads in flight. The occupancy of the GPU device is kept constant. The pusher kernel can be expected to be limited by memory access latencies. A larger number of active blocks could help hiding such latencies. However, such an increase of device-side parallelism is not possible as the large number of registers used by the kernel causes the number of available registers to become exhausted.

5.2 Clock Rates

Another parameter of GPU architectures are the clock rates with which the GPU operates. The P100 device on JURON can operate with graphics clocks between 544 MHz and 1480 MHz (the memory clock is fixed at 715 MHz); the K40 device operates between 666 MHz and 875 MHz at a memory frequency of 3004 MHz; the K80 can run with 562 MHz to 875 MHz at a slightly lower memory frequency, compared to the K40, of 2505 MHz.

Building upon the performance model of Eq. 1 the following relation can be formulated to model the effect of different GPU clock rates ($\mathcal{C}$) on effective bandwidths (β):

$$\beta(\mathcal{C}) = \gamma + \delta\mathcal{C} \tag{2}$$

As before, γ and δ are fit parameters.

To obtain δ and γ for one device, each effective bandwidth β for a possible clock rate is obtained per Eq. 1 – in each case the runtime of the GPU pusher kernel is measured for different amounts of exchanged information (*number of*

[4] Although the value of 720 GB/s is the design value of the P100, it might be different from a practical achievable bandwidth. Indeed, we measure a bandwidth of about 520 GB/s for the four mini-benchmarks of the STREAM benchmark. Using this as a reference value, the pusher kernel manages to use slightly more than 50% of this empirically determined bandwidth limit.

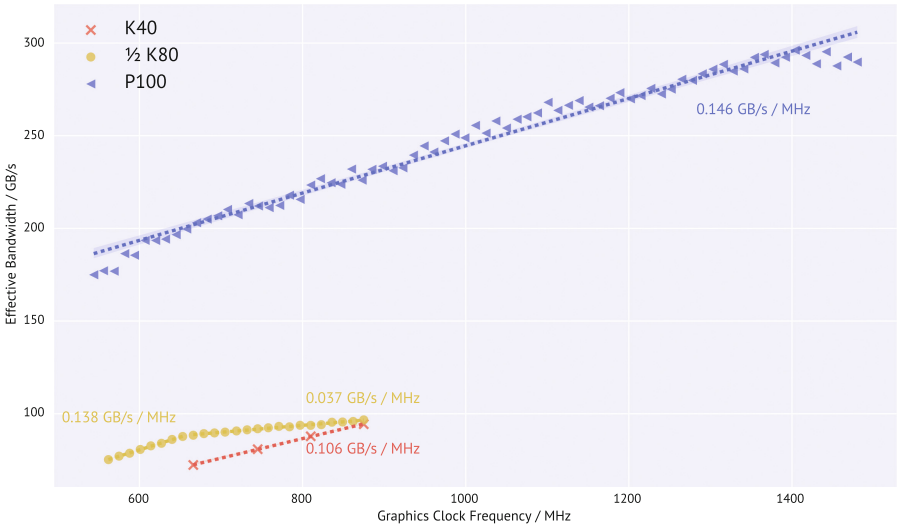

Fig. 4. Effective bandwidths from the performance model (see Eq. 1) for different graphics clock frequency. Shown are three GPU devices: K40, ½ K80, and P100. Superimposed are δ values of linear fits to the respective measurements using Eq. 2.

particles), and the slope of the linear fit noted. The resulting measurements are shown in Fig. 4 for the three GPUs.

The P100 covers a large range of possible clock rates, with the maximal values leading to bandwidths close to 300 GB/s. But also for frequencies below 875 MHz on which the K40 and K80 can operate on as well, the P100 surpasses the older devices vastly. As already determined in Sect. 5.1, the pusher benefits greatly from the increased memory bandwidth, and does so independently from the clock rate. Additionally, the Pascal architecture increases the number of multiprocessors over the previous Kepler systems; JuSPIC is capable of exploiting this as well. In total, the dependence of the bandwidth on the graphics clock is $\delta = 0.146\,(\mathrm{GB/s})/(\mathrm{MHz})$

The chips on K40 and on K80 are very similar, which can be seen in the values for the largest clock frequencies – they are close to identical. The performance is different for smaller clock rates: The bandwidth is reduced less for K80 devices. The chip seems to operate more efficiently. A distinct feature is visible for the K80: The distribution has two parts. For lower clock rates the effective bandwidth increases faster (0.138 (GB/s)/(MHz)) than for higher clock rates (0.037 (GB/s)/(MHz)). The kink in the curve marks the position where nearly the highest computing performance is reached – for further increase in clock rate only little performance is gained. The model parameter for the K40 is 0.106 (GB/s)/(MHz).

Judging from the δ parameters, it can be seen that the P100 device not only has the absolute best performance but also the largest increase in effective bandwidth with each step in clock frequency.

6 Related Work

Various PIC codes have been ported to GPU. The PSC code, on which JuSPIC is based, was later reimplemented in C and ported to GPUs [11]. More specifically, the code has been ported to the Titan supercomputer at Oak Ridge National Lab, a Cray XK7 system comprising just over 18 000 nodes with 1 GPU each. The same system was used to demonstrate extreme scalability of the PIConGPU, a PIC code specifically developed for GPU acceleration [9]. While PSC and PIConGPU are full production codes, other efforts for GPU porting have been performed using proof-of-concept codes [10,16,19].

As the Minksy platform is relatively new, not much work has been published exploring the performance of scientific applications on this platform. Various publications investigated the performance using the precursor platform where NVDIA K40 or K80 GPUs where attached to POWER8 processors via a PCIe GEN3 link. This included, e.g., evaluation of applications based on the Finite-Difference Time-Domain (FDTD) method [4], based on the Density Function Theory (DFT) method [5], or molecular dynamics simulations [20].

7 Summary and Conclusions

In this paper, we reported on our progress of accelerating the plasma physics PIC code JuSPIC with GPU devices.

A heterogeneous approach of employed programming models is chosen. We use OpenACC for data movement and simple kernels operating on three-dimensional fields. OpenACC offers the ability of creating portable and *backwards*-compatible code with only few annotating compiler directives.

For the most compute-intensive routine, the particle pusher, we use CUDA Fortran since earlier versions of PGI's OpenACC compiler were not able to generate efficient code for the original data structures. Compatibility to systems without GPUs is achieved by guarding the CUDA Fortran code with pre-processor directives. For the CUDA Fortran kernel, we evaluate different data layouts. The Structure-of-Array approach is fastest, providing best performance when moving data between host and device and smallest kernel runtime. In the future we want to implement pinned host data for this case, learning from the benefits of *Experiment 2*.

In the process of analysing the individual stages of the CUDA Fortran part we notice unexpected high runtimes for pre- and post-processing steps. In these steps the linked list of particles is copied to and from simple Fortran fields (to be processed on the GPU). Using a boiled-down benchmark we investigate this peculiarity and determine the performance issue to be a `malloc` call which is issued by the OpenMPI version shipped with the POWER version of the

PGI compiler. Although we find workarounds, a possible mitigation is yet to be applied to JuSPIC.

Moving on, we study the performance of the kernel of the pusher with different number of particles in the scope of a simple information exchange model. Four different GPU devices are investigated with the P100 of JURON providing by far the best performance. The GPU provides an effective bandwidth of 285 GB/s with its default clock setting. Subsequently, the performance model is adapted to incorporate different GPU clock frequencies. The P100 provides the most efficient scaling also in this case. An interesting additional investigation objective for the future is the incorporation of energy measurements – which device takes the least energy to come to a solution?

JuSPIC is a good fit for the new Pascal GPU architecture, benefiting well from the increased memory bandwidth. Currently, only the part of the pusher is ported to the GPU. We expect the performance gain from the GPU-accelerated version to be significant, once also the reducer is ported to the GPU. The data movements from and to the devices can be omitted in this case, reducing the overhead. Once the single-node version is accelerated, a next step will be Multi-GPU usage together with MPI. Currently, the OpenMP statements available in JuSPIC are ignored for the GPU version, to solely focus on this part of the acceleration and prevent race conditions. Once the GPU version is stable, we should ensure leveraging all possibilities of potential parallelism and enable OpenMP again.

Not only the GPU version of JuSPIC is currently developed, the code itself is progressing further. Different data layouts are being investigated to possibly remove storing particle data in linked lists, simplifying coalesced data handling. Effective load-balancing using space-filling curves is also currently studied.

Acknowledgements. This work has been carried out in the context of the *POWER Acceleration and Design Center*, a joined project between IBM, Forschungszentrum Jülich and NVIDIA, as well as the *NVIDIA Application Lab at Jülich*, a joined project between Forschungszentrum Jülich and NVIDIA. We acknowledge the support from Jiri Kraus (NVIDIA) and various helpful discussions with him. Research leading to these results has (in parts) been carried out on the Human Brain Project PCP Pilot Systems at the Juelich Supercomputing Centre, which received co-funding from the European Union (Grant Agreement no. 604102).

References

1. The Jülich Scalable Particle-in-Cell code, JuSPIC, http://www.fz-juelich.de/ias/jsc/juspic/
2. JuSPIC in the High-Q Club, http://www.fz-juelich.de/ias/jsc/EN/Expertise/High-Q-Club/JuSPIC/_node.html
3. JuSPIC Source Code Repository, https://trac.version.fz-juelich.de/juspic
4. Baumeister, P.F., Hater, T., Kraus, J., Pleiter, D., Wahl, P.: A performance model for GPU-accelerated FDTD applications. In: 2015 IEEE 22nd International Conference on High Performance Computing (HiPC), pp. 185–193, December 2015

5. Baumeister, P.F., Bornemann, M., Bühler, M., Hater, T., Krill, B., Pleiter, D., Zeller, R.: Addressing materials science challenges using GPU-accelerated POWER8 nodes. In: Dutot, P.-F., Trystram, D. (eds.) Euro-Par 2016. LNCS, vol. 9833, pp. 77–89. Springer, Cham (2016). doi:10.1007/978-3-319-43659-3_6
6. Birdsall, C.K., Langdon, A.B.: Plasma Physics via Computer Simulation. Series in Plasma Physics. Taylor & Francis, New York (2005)
7. Bonitz, M., Semkat, D. (eds.): Introduction to Computational Methods in Many Body Physics. Rinton Press, Princeton (2006)
8. Brömmel, D., Gibbon, P., Garcia, M., Lopez, V., Marjanovic, V., Labarta, J.: Experience with the MPI/STARSS programming model on a large production code. In: International Conference on Parallel Computing: Accelerating Computational Science and Engineering (CSE). Advances in Parallel Computing, vol. 25, pp. 357–366, Munich, Germany, 10–13 September 2013. IOS Press (2014)
9. Bussmann, M., Burau, H., Cowan, T.E., Debus, A., Huebl, A., Juckeland, G., Kluge, T., Nagel, W.E., Pausch, R., Schmitt, F., Schramm, U., Schuchart, J., Widera, R.: Radiative signature of the relativistic Kelvin-Helmholtz instability. In: 2013 SC - International Conference for High Performance Computing, Networking, Storage and Analysis (SC), pp. 1–12, November 2013
10. Decyk, V.K., Singh, T.V.: Adaptable particle-in-cell algorithms for graphical processing units. Comput. Phys. Commun. **182**(3), 641–648 (2011), http://www.sciencedirect.com/science/article/pii/S0010465510004558
11. Germaschewski, K., Fox, W., Abbott, S., Ahmadi, N., Maynard, K., Wang, L., Ruhl, H., Bhattacharjee, A.: The Plasma Simulation Code: A modern particle-in-cell code with load-balancing and GPU support. ArXiv e-prints, October 2013
12. Gopal, A., Herzer, S., Schmidt, A., Singh, P., Reinhard, A., Ziegler, W., Brömmel, D., Karmakar, A., Gibbon, P., Dillner, U., May, T., Meyer, H.G., Paulus, G.G.: Observation of gigawatt-class THz pulses from a compact laser-driven particle accelerator. Phys. Rev. Lett. **111**(7), 074802 (2013)
13. Harlow, F.H.: The particle-in-cell method for numerical solution of problems in fluid dynamics, March 1962, http://www.osti.gov/scitech/servlets/purl/4769185
14. Herten, A., Pleiter, D., Brömmel, D.: Accelerating Plasma Physics with GPUs (Poster). Tech. rep., GPU Technology Conference (2017)
15. Hockney, R.W., Eastwood, J.W.: Computer simulation using particles. Institute of Physics, Bristol (1988) (English)
16. Kong, X., Huang, M.C., Ren, C., Decyk, V.K.: Particle-in-cell simulations with charge-conserving current deposition on graphic processing units. J. Comput. Phys. **230**(4), 1676–1685 (2011), http://www.sciencedirect.com/science/article/pii/S0021999110006479
17. Mucci, P., ICL Team, T.: PAPI, the performance application programming interface, http://icl.utk.edu/papi/
18. Mucci, P.J., Browne, S., Deane, C., Ho, G.: PAPI: a portable interface to hardware performance counters. In. Proceedings of the Department of Defense HPCMP Users Group Conference, pp. 7–10 (1999)
19. Stantchev, G., Dorland, W., Gumerov, N.: Fast parallel particle-to-grid interpolation for plasma PIC simulations on the GPU. J. Parallel Distrib. Comput. **68**(10), 1339–1349 (2008), http://dx.doi.org/10.1016/j.jpdc.2008.05.009
20. Weber, V., Malossi, A.C.I., Tavernelli, I., Laino, T., Bekas, C., Modani, M., Wilner, N., Heller, T., Curioni, A.: First experiences with *ab initio* molecular dynamics on OpenPOWER: the case of CPMD. In: Taufer, M., Mohr, B., Kunkel, J.M. (eds.) ISC High Performance 2016. LNCS, vol. 9945, pp. 228–234. Springer, Cham (2016). doi:10.1007/978-3-319-46079-6_16