GPU-Accelerated Particle-in-cell Code on Minsky

IWOPH17, ISC, Frankfurt a. M.

Andreas Herten, Forschungszentrum Jülich, 22 July 2017 *Handout Version*

Outline

Contributions *TL;DR*

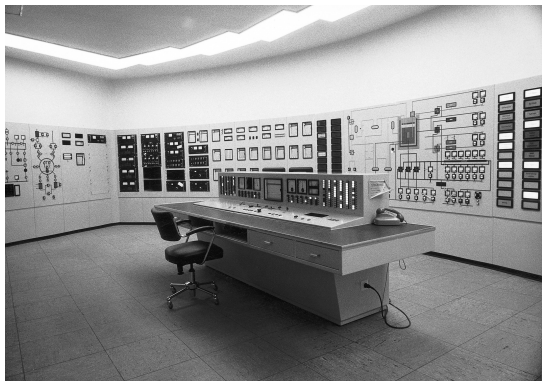
- PiC Code to GPU (partly)
- OpenACC, CUDA Fortran
- Data layout benchmarks on Minsky (POWER8*NVL*, P100)
- Peculiarities with PGI compiler on POWER
- Performance Model

Jülich Supercomputing Centre

Part of Forschungszentrum Jülich

JÜLICH FORSCHUNGSZENTRUM

- Forschungszentrum Jülich
  - One of Europe's largest research centers (≈6000 employees)
  - Energy, environmental sciences, health, information technology
- Jülich Supercomputing Centre
  - Two Top 500 supercomputers (JUQUEEN: #21, JURECA: #80)
  - NVIDIA Application Lab
  - POWER Acceleration and Design Centre



JÜLICH APPLICATION LAB NVIDIA

- Jülich Supercomputing Centre is part of Institute for Advanced Simulation of Forschungszentrum Jülich. Apart from operating supercomputers and managing access to them, we research in new computer technologies and support applications running on the machines.
- NVIDIA Application Lab is a collaboration with NVIDIA: Andreas from JSC + Jiri Kraus from NVIDIA. We consult scientific applications regarding GPGPU computing.
- POWER Acceleration and Design Centre is a collaboration with NVIDIA and IBM to investigate OpenPOWER infrastructure for HPC needs with applications from different scientific fields.

*Super*computers Involved

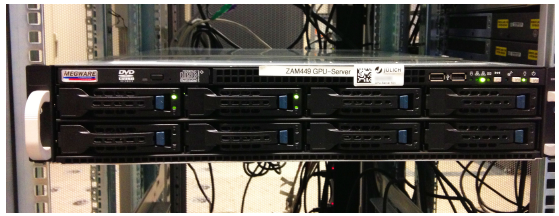
JÜLICH FORSCHUNGSZENTRUM

JURON

- Human Brain Project prototype
- 18 nodes with IBM POWER8*NVL* CPUs (2×10 cores)
- Per Node: 4 NVIDIA Tesla P100 cards, connected via NVLink.
- GPU: 0.38 PFLOP/s peak performance
- NVME

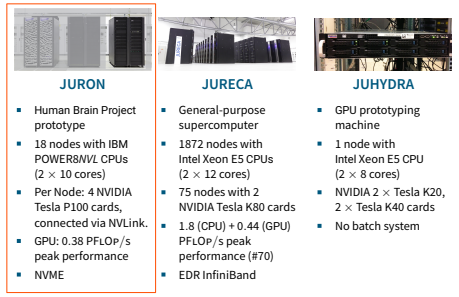
JURECA

- General-purpose supercomputer
- 1872 nodes with Intel Xeon E5 CPUs (2×12 cores)
- 75 nodes with 2 NVIDIA Tesla K80 cards
- 1.8 (CPU) + 0.44 (GPU) PFLOP/s peak performance (#70)
- EDR InfiniBand



JUHYDRA

- GPU prototyping machine
- 1 node with Intel Xeon E5 CPU (2×8 cores)
- NVIDIA $2 \times$ Tesla K20, $2 \times$ Tesla K40 cards
- No batch system



Within context of the paper at hand we use three systems: JURON, JURECA, and JUHYDRA, where the latter two are only used for comparisons.

JURON	Large OpenPOWER system which is benchmarked, based on IBM S822LC; 1 processor and 2 GPUs connected in ring topology; with Centaur memory
JURECA	Multi-purpose supercomputer used regularly for JuSPIC
JUHYDRA	Very small benchmarking system available in-house to study GPUs with small turn-around times

If not stated differently, JuSPIC is run on JURON.

JÜLICH
FORSCHUNGSZENTRUM

JuSPIC

JuSPIC

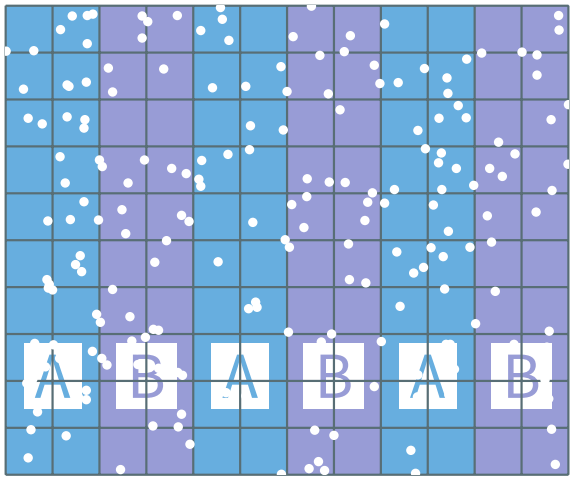
A scalable Particle-in-Cell plasma physics code

JÜLICH FORSCHUNGSZENTRUM

- Based on PSC by H. Ruhl
- Laser-plasma interaction
- 3D electromagnetic PiC code
- Finite-Difference Time-Domain scheme
- Cartesian geometry, arbitrary number of particle species
- Scales to full Blue Gene/Q system JUQUEEN

HiQ

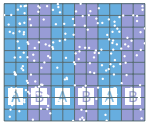
- Modern Fortran, Open Source
- Distributed with MPI in tiles
- CPU-parallelized with OpenMP



JuSPIC

A scalable Particle-in-Cell plasma physics code

- Based on PSC by H. Ruhl
- Laser-plasma interaction
- 3D electromagnetic PiC code
- Finite-Difference Time-Domain scheme
- Cartesian geometry, arbitrary number of particle species
- Scales to full Blue Gene/Q system JUQUEEN
- Modern Fortran, Open Source
- Distributed with MPI in tiles
- CPU-parallelized with OpenMP



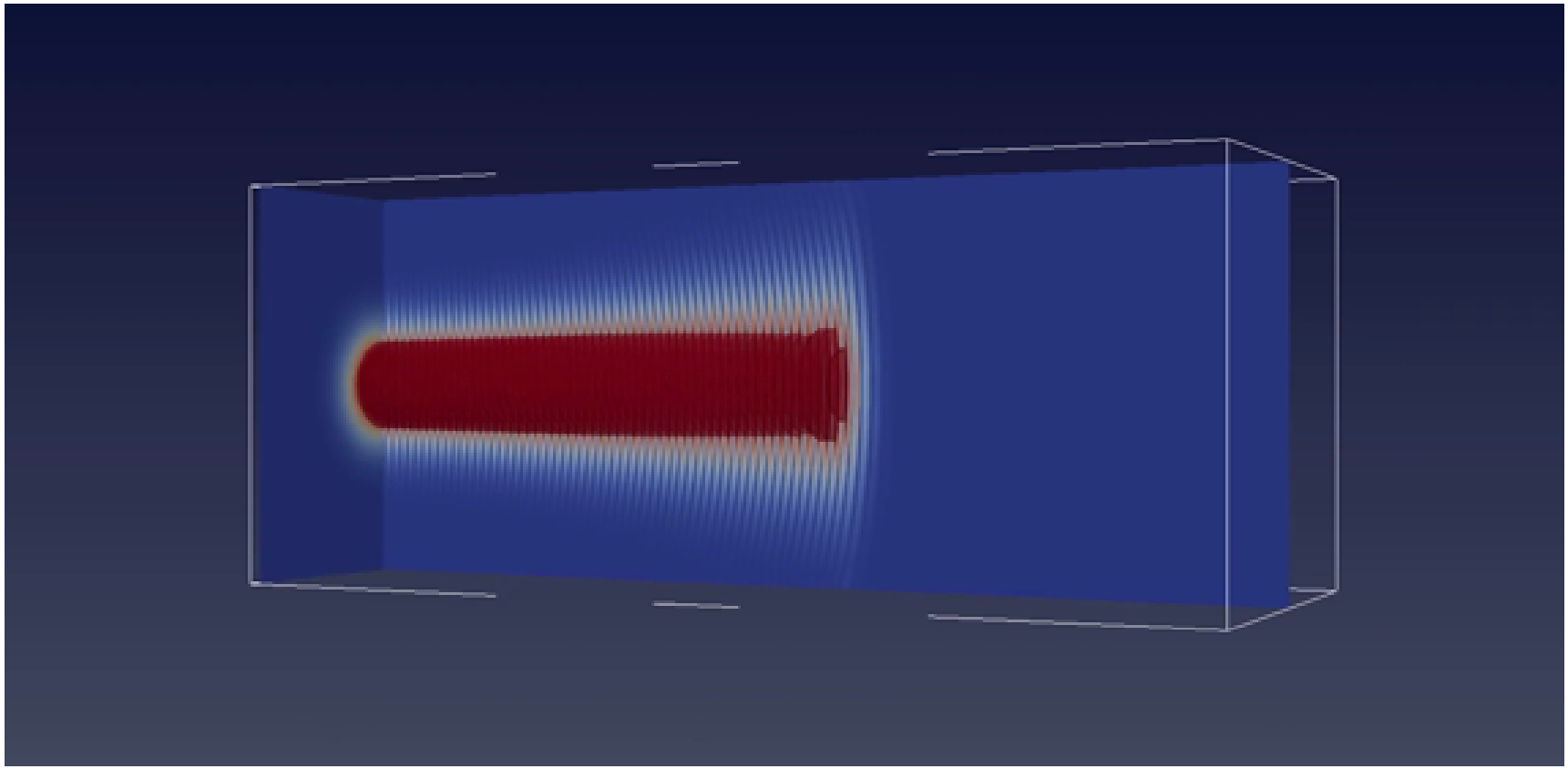
HiQ

- FDTD: Decoupling of electromagnetic differential equation; solving in leap-frog manner (with $t_{1/2}$ time steps)
- MPI and OpenMP both disable for investigations at hand
- But consciously programmed to be compatible

Sample Simulation

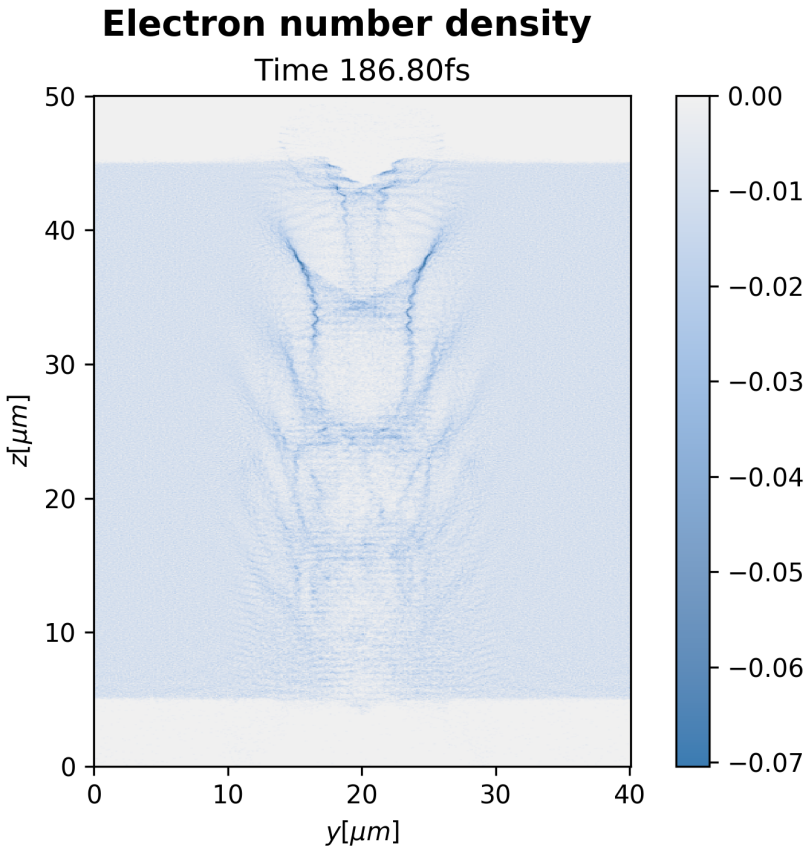
Visualizing different quantities

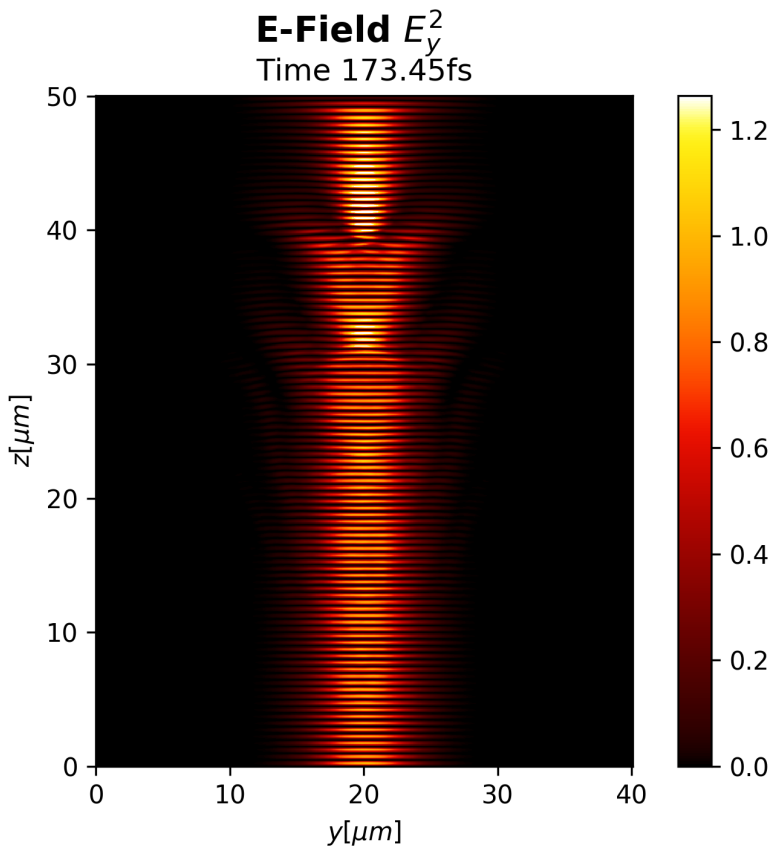
JÜLICH FORSCHUNGSZENTRUM

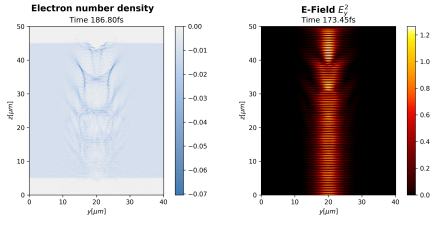


Sample Simulation

Visualizing different quantities

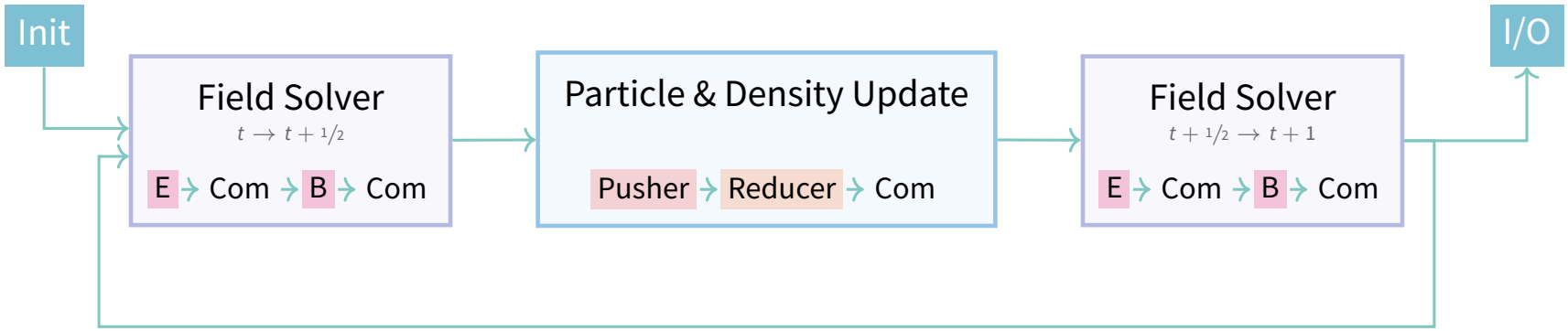




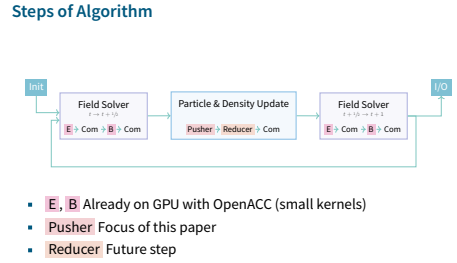


- JuSPIC allows for inspection of all quantities at any point in time due to I/O
- Electron number density is actually electron charge number density (hence negativity)

Steps of Algorithm



- E , B Already on GPU with OpenACC (small kernels)
- Pusher Focus of this paper
- Reducer Future step



- *Init* initializes all data structures and configurations (via config file)
- Leapfrog method (finite difference time domain): fields solved in half-time steps
- Field solvers already on GPU (OpenACC)
- Update step most compute-intensive part (because of large number of particles)
- Current focus: Pusher (Update quasi-particle properties with info from field grid)
- Next up: Reducer (reduce particle info back to field; irregular access pattern)
- Communications between different MPI ranks (MPI disabled here, but handled correctly)

JÜLICH
FORSCHUNGSZENTRUM

Acceleration for GPUs

JÜLICH
FORSCHUNGSZENTRUM

Acceleration for GPUs

OpenACC

OpenACC in JuSPIC

A long story

JÜLICH FORSCHUNGSZENTRUM

- Field solvers use OpenACC (simple code)

```
!$acc kernels loop collapse(3) present(e,b,ji)
do i3=i3mn-1,i3mx+1
   do i2=i2mn-1,i2mx+1
      do i1=i1mn-1,i1mx+1
         e(i1,i2,i3)%X=e(i1,i2,i3)%X
         ! etc
```

- Data movement with OpenACC (incl. resident parts)
- But Pusher no easy feat

- JuSPIC is (also) benchmark code
- Should run on as many architecture as possible without changes
- OpenACC perfect match?!
- In principle: Yes…, but Pusher!

OpenACC in JuSPIC
A long story

- Field solvers use OpenACC (simple code)

```
!$acc kernels loop collapse(3) present(e,b,ji)
do i3=i3mn-1,i3mx+1
   do i2=i2mn-1,i2mx+1
      do i1=i1mn-1,i1mx+1
         e(i1,i2,i3)%X=e(i1,i2,i3)%X
         ! etc
```

- Data movement with OpenACC (incl. resident parts)
- But Pusher no easy feat

Accelerating Plasma Physics with GPUs

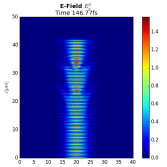
JuSPIC with OpenACC and CUDA Fortran

Andreas Herten, Dirk Pleiter, Dirk Brömmel
Jülich Supercomputing Centre

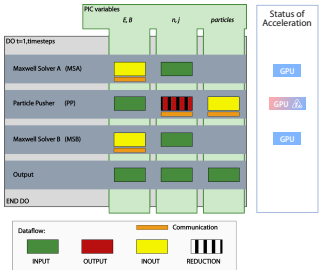
JuSPIC

Jülich Scalable Particle-in-Cell Code

- Based on plasma simulation code *PSC* (by H. Ruhl)
- 3D electromagnetic Particle-in-Cell code
- Solves relativistic Vlasov equation, coupled to Maxwell equations in finite-difference time-domain scheme
- Cartesian geometry; arbitrary number of particle species

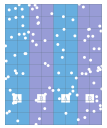


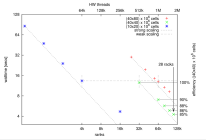
Workflow



Techniques

- Modern Fortran
- Fully distributed with MPI
  - Domain decomposition: tiles
- CPU-parallelized with OpenMP
  - Local decomposition: slices
  - A, B processed independently
- Scales to full JUQUEEN supercomputer

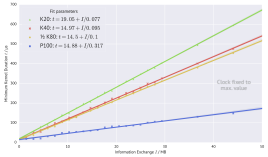




HiQ

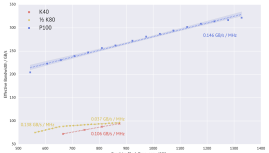
Performance Model

Based on information exchange: $t(N_{Part}) = a + l(N_{Part})/\beta$



Kernel duration vs. information exchange

- Information exchanged for kernel
  → Is lower limit of exploited bandwidth
- Effective bandwidth: K80 – 100 GB/s; P100 – 317 GB/s
- GPU kernel possibly latency-limited (many registers)



Effective bandwidths vs. clock frequencies for K40, K80, P100

- K80: Two regions (*left: performance depending on clock; right: nearly constant*)
- P100: JuSPIC benefits from new GPU design

Status of Porting and Acceleration

Particle Pusher

Three parts:
- Solve Maxwell equations with OpenACC *(not shown here)*
- Update of particle position & momentum (*pusher*)
- Update of densities (*reduction*)

GPU ✓ GPU GPU

CPU Version
- Single core (OpenMP disabled)
- Original data structure (linked list) temporarily moved to array
  → **overhead**

Fast OpenACC
- **Rewrite of entire computing kernel necessary!**
- Few Fortran features used (*arrays…*)

Initial OpenACC Port
- **Not running!**
- Breaks at first encounter

Working OpenACC Port
- Two changes necessary
  - Unroll some array operations
  - Limit number of gang/vector (***slow!***)

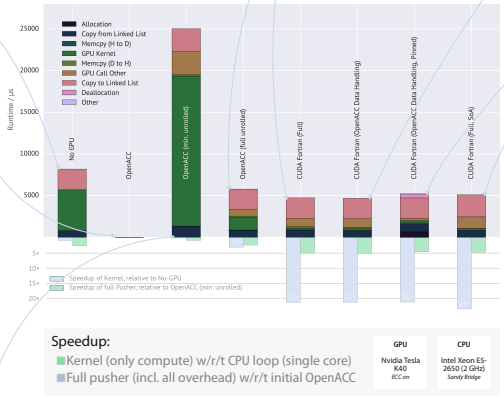
→ Fortran programming style and complex kernel challenging for OpenACC compiler

CUDA Fortran
- Translation to CUDA Fortran kernel
- Helper data (scalars, 3D vectors) handled by OpenACC
- Particle pos., mom. via CUDA
- GPU-compatible through preprocessor guards

CUDA Fortran + OpenACC
- All data (incl. large arrays) handled by OpenACC
- Few code changes necessary

***FUDA* + Pinned OpenACC**
- Pinned host data
- Faster data staging

CUDA Fortran, SoA
- Structure-of-Array data type for coalesced memory access
- Allocated once, resized dynamically
- Speedup single CPU: 24×



Speedup:
- Kernel (only compute) w/r/t CPU loop (single core)
- Full pusher (incl. all overhead) w/r/t initial OpenACC

GPU	CPU
Nvidia Tesla K40 *ECC on*	Intel Xeon E5-2650 (2 GHz) *Sandy Bridge*

OpenACC and Fortran

- Support through PGI compiler
- Well-supported, example:

```
!$acc parallel loop
private(pp,root,qi,mi,wi)
present(e,b)
do i=1, n
  ! …
end do
!$acc end parallel
```

- JuSPIC: Many issues during parallelization
- Structured datatypes and array operations challenging for compiler

→ Many manual code adaptions

- Why not use CUDA Fortran?
- Portable with preprocessor guards!

```
#ifdef _CUDA
i = blockDim%x * blockIdx%x +
threadIdx%x
#else
do i=0,N
#endif
! …
```

Conclusion & Outlook

Conclusion
- First progress made in GPU-acceleration of JuSPIC
- Hybrid code: OpenACC and CUDA Fortran
- Changes in data layout necessary (expensive!)
- Benefit from P100 architecture

Outlook
- Reduction on GPU
- Minimization of host/devices copies
- Lowering of overhead of data layout transformations
- Evaluate data layout change for rest of JuSPIC
- Parallelization on slice / tile level
- Parallelization on multiple GPUs

JÜLICH APPLICATION LAB NVIDIA

JÜLICH FORSCHUNGSZENTRUM

JÜLICH
FORSCHUNGSZENTRUM

Acceleration for GPUs

CUDA Fortran

Introduction to CUDA Fortran

It's like CUDA C/C++,… but for Fortran

JÜLICH FORSCHUNGSZENTRUM

- Available in PGI Fortran compiler
- Adds CUDA extensions to Fortran
- Examples (from JuSPIC):
  - Define device function along-side host function

```
type(particle_type), dimension(slice(1)%n) ::
↪   list_of_particles, list_of_particles_d
attributes(device) :: list_of_particles_d
```

  - Copy to device

```
list_of_particles_d = list_of_particles
```

  - Define kernel

```
attributes(global) subroutine gpupusher(list_of_particles, ...)
```

  - Call kernel

```
call gpupusher<<<dim3(nBlocks, 1, 1), dim3(nThreads, 1,
↪   1)>>>(list_of_particles_d, ...)
```

CUDA Fortran Portability

Not as portable as OpenACC, but it's alright

JÜLICH FORSCHUNGSZENTRUM

- CUDA Fortran: more powerful approach
- Portability suffers…
- … but can be mitigated!

1. Use OpenACC as much as possible, e.g. for data movements
   OpenACC mixes well together with CUDA Fortran

```
!$acc enter data copyin(list_of_particles, ...)
```

2. Use pre-processor directives for rest

```
#ifdef _CUDA
  i = blockDim%x * (blockIdx%x - 1) + threadIdx%x
#else
  do i = lbound(a, 1), ubound(a, 1)
#endif
```

CUDA Fortran Portability
Not as portable as OpenACC, but it's alright

- CUDA Fortran: more powerful approach
- Portability suffers…
- … but can be mitigated!

1. Use OpenACC as much as possible, e.g. for data movements
   OpenACC mixes well together with CUDA Fortran

```
!$acc enter data copyin(list_of_particles, ...)
```

2. Use pre-processor directives for rest

```
#ifdef _CUDA
  i = blockDim%x * (blockIdx%x - 1) + threadIdx%x
#else
  do i = lbound(a, 1), ubound(a, 1)
#endif
```

- OpenACC Fortran (or C/C++) interoperability
  - `host_data use_device` to use use device pointer also on host side
  - Use host-side data containers for kernels (because OpenACC automatically translates)
- Apart from `_CUDA`, there's also `_OPENACC`

JÜLICH
FORSCHUNGSZENTRUM

Acceleration for GPUs

Data Layout Analysis

Strategies for Data Layout

Because data is not solely data

JÜLICH FORSCHUNGSZENTRUM

- Benchmark different data layouts and transfer strategies
- Sub-parts of Pusher:

$\sum$	*Everything*
Allocate	Allocate host-side data structures
LL2F	Convert linked-list data structure to field
H2D	Copy data from host to device
Kernel	Run kernel
D2H	Copy data from device to host
Other	Left-over time (synchronization, etc.)
F2LL	Copy flat field back to linked list

- Benchmarking on **JURON**

Data Layout Experiments

Description of experiments

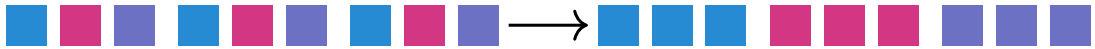
JÜLICH FORSCHUNGSZENTRUM

Initial All particles stored in single field, one particle after another; data copied to/from GPU with Fortran *(baseline)*

Exp 1 As *Initial*, but data copied with OpenACC *copy* directives

Exp 2 As *Exp 1*, but data copied from pinned host memory

SoA Data copied with Fortran, but instead of one field with all particle data, one field for each spatial and momentum component for particles



in µs	$\sum$	Allocate	LL2F	H2D	Kernel	D2H	Others	F2LL
Initial	8040	–	567	82	84	62	350	6885
Exp 1	10435	–	353	80	82	91	380	9440
Exp 2	9695	564	527	79	83	72	108	7973
SoA	7811	1	844	66	77	53	376	6386

Data Layout Experiments

Discussion of results

JÜLICH FORSCHUNGSZENTRUM

in µs	$\sum$	Allocate	LL2F	H2D	Kernel	D2H	Others	F2LL
Initial	8040	–	567	82	84	62	350	6885
Exp 1	10435	–	353	80	82	91	380	**9440**
Exp 2	9695	**564**	527	79	83	**72**	**108**	7973
SoA	7811	1	**844**	**66**	**77**	**53**	376	6386

- **SoA**: fastest, looking (also) at raw GPU runtimes – but slowest for change of data structures (six fields vs. one)
- **Exp 2**: least overhead; pinned memory allows for direct data access – but allocation overhead is not fully resolved
- **Exp 1**: also ok for raw GPU times, but large F2LL overhead *(more on that later)*

Data Layout Experiments

Discussion of results

in µs	∑	Allocate	LL2F	H2D	Kernel	D2H	Others	F2LL
Initial	8040	–	567	82	84	62	350	6885
Exp 1	10435	–	353	80	82	91	380	**9440**
Exp 2	9695	**564**	527	79	83	**72**	**108**	7973
SoA	7811	1	**844**	**66**	**77**	**53**	376	6386

- SoA: fastest, looking (also) at raw GPU runtimes – but slowest for change of data structures (six fields vs. one)
- Exp 2: least overhead; pinned memory allows for direct data access – but allocation overhead is not fully resolved
- Exp 1: also ok for raw GPU times, but large F2LL overhead *(more on that later)*

- SoA: GPU performance good as expected (coalesced memory loads); overhead during LL2F because of accessing six fields at different memory locations
- AFAIK OpenACC stages data always implicitly into a pinned host buffer before H2D, hence the overhead
- I don’t know why F2LL is so much higher for Exp 1 as for Exp 2

Data Layout Experiments

Architecture Comparison

in µs	$\sum$	Allocate	LL2F	H2D	Kernel	D2H	Others	F2LL
JURON								
Initial	8040	–	567	82	84	62	350	6885
Exp 1	10435					91	380	9440
Exp 2	9695					72	108	7973
SoA	7811				77	53	376	6386
JUHYDRA								
Initial	4956	–	908	267	229	208	736	2600
Exp 1	4687	–	764	232	229	198	804	2455
Exp 2	5328	577	1027	224	230	192	23	2651
SoA	4880	1	786	204	208	173	827	2674

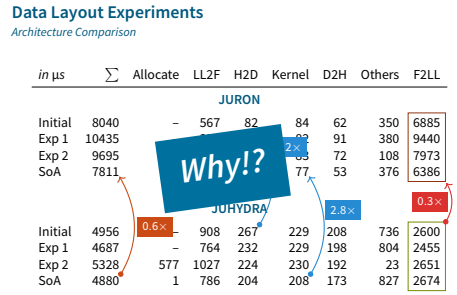
Why!?

2×

2.8×

0.6×

0.3×



- JURON (with P100) about 3× faster than JUHYDRA (with K40), looking at core-GPU performance
- But overall time is slower!
- Reason: F2LL – but why?
- Further improvements: do not re-allocate pinned memory for every algorithm step; move data regions up further; SoA + pinned; hopeful when next part (Reducer) of algorithm is also on GPU

JÜLICH
FORSCHUNGSZENTRUM

Acceleration for GPUs

Data Layout Conversion

Conversion of Data Layouts

Why is F2LL so slow?

JÜLICH FORSCHUNGSZENTRUM

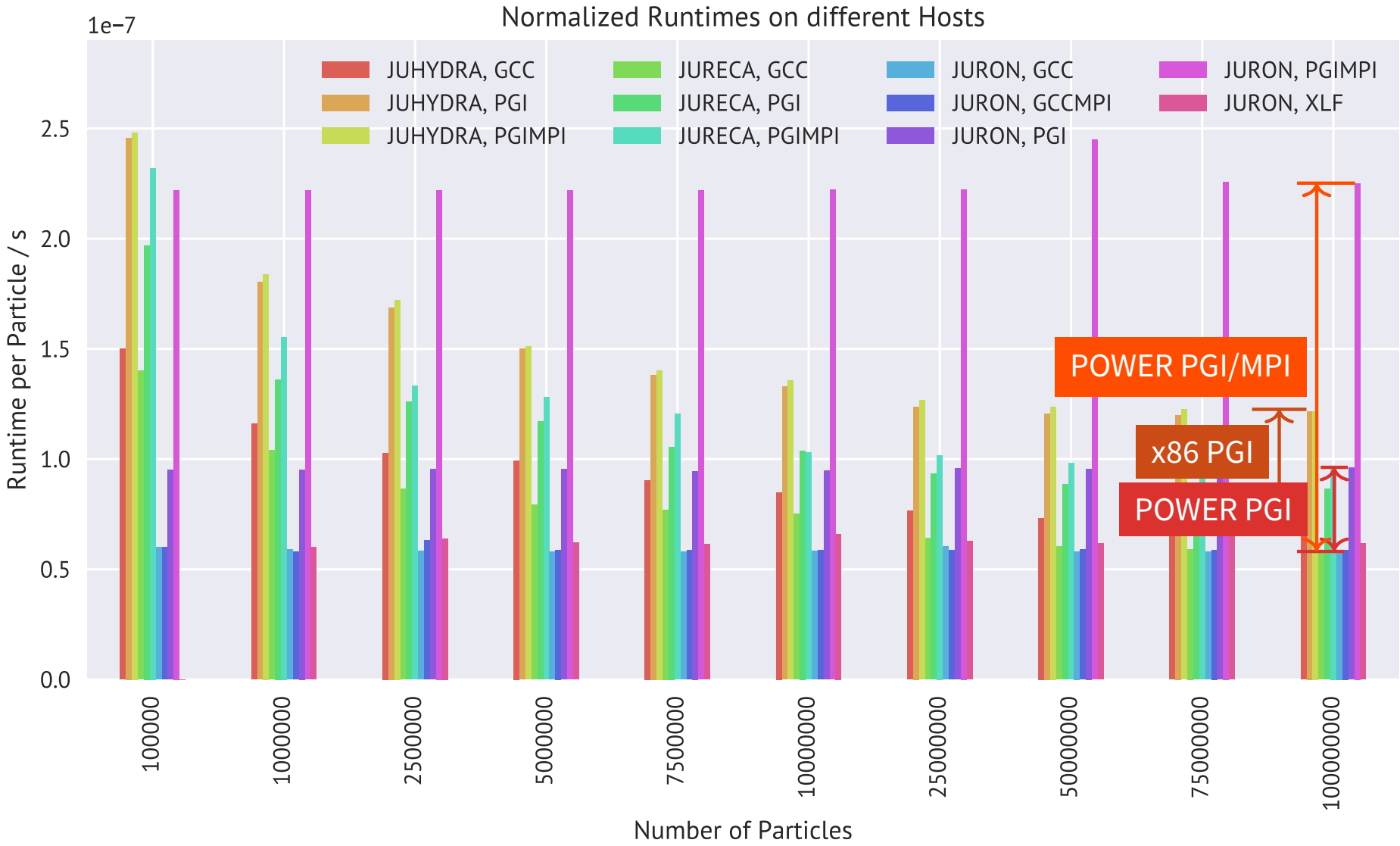
- Parts of F2LL
  - Kill old linked list of particles[1]
  - Initialize new, empty linked list of particles
  - Loop through field(s) of particle information…
  - … add each particle to linked list, update pointers
- `add_one_to_list`

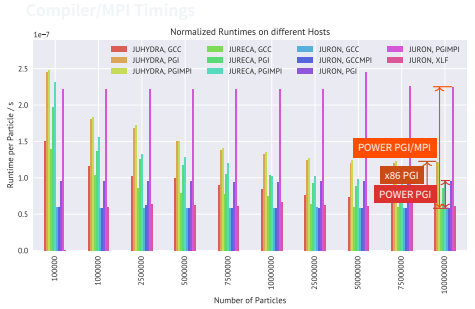
```
allocate(list%tail%next)
nullify(list%tail%next%next)
list%tail%next%particle = particle
list%tail => list%tail%next
```

⇒ ***Benchmark***

[1] Start with first particle, progress along links, remove each particle

Compiler/MPI Timings





- PGI consistently 30 % to 60 % slower than GCC
- PGIMPI 2× slower than PGI without MPI!
- See appendix for `remove_from_list`

Compiler Investigation

Is MPI Slow? And, by the way, which MPI!?

JÜLICH FORSCHUNGSZENTRUM

- *PGIMPI*: MPI version shipped with PGI
- Not actively used in GPU version of JuSPIC, but in future
- `add_one_to_list` benchmark does not use MPI at all!
  Replacing `pgfortran` by `mpifort` leads to performance decrease

→ ***Benchmark*** compilers – with PAPI [3] instrumentation

System	JURON						JUHYDRA	
Compiler	GCC	GCCMPI	PGI	PGIMPI	PGIMPI*	XLF	PGI	PGIMPI
Time pP/ns	36	37	46	154	48	41	32	32
Instructions pP	121	121	243	462	243	121	210	210

*See **appendix** for some more counters*

Compiler Investigation
Is MPI Slow? And, by the way, which MPI!?

- *PGIMPI*: MPI version shipped with PGI
- Not actively used in GPU version of JuSPIC, but in future
- `add_one_to_list` benchmark does not use MPI at all! Replacing `pgfortran` by `mpifort` leads to performance decrease

→ ***Benchmark*** compilers – with PAPI [3] instrumentation

System	JURON						JUHYDRA	
Compiler	GCC	GCCMPI	PGI	PGIMPI	PGIMPI*	XLF	PGI	PGIMPI
Time pP/ns	36	37	46	154	48	41	32	32
Instructions pP	121	121	243	462	243	121	210	210

*See **appendix** for some more counters*

- PGIMPI*: Custom OpenMPI 2.0 version (for the benchmark test case, JuSPIC not tested)
- GCC: 5.4.0
- PGI: 16.10 (JURON), 16.3 (JUHYDRA)
- GCCMPI: OpenMPI 2.0.2
- PGIMPI: OpenMPI 1.10.2 (JURON), OpenMPI 1.8.1 (JUHYDRA)
- XLF: 16.1.0

- PAPI counter: $|\text{PAPI\_TOT\_INS}| \rightarrow |\text{PM\_INST\_CMPL}|$ (JURON), $|\text{INSTRUCTION\_RETIRED}|$ (JUHYDRA)
- *pP*: per Particle

Further Investigation/Mitigation

JÜLICH FORSCHUNGSZENTRUM

- MPI version shipped with PGI on POWER is slow, because it issues many instructions
- Further study: Identical assembly code generated as MPI-less version…
- … but includes call to `malloc()`!
- Different libraries linked for PGI and PGIMPI cases!
- `LD_PRELOAD=/lib64/libc.so.6` solves problem!

⇒ Slow MPI-aware `malloc()`?

- Mitigation
  - **Bug reported**
  - For now: consider as *anomalous* overhead

Mitigation

- Wait for bug being fixed
- Thoroughly test custom MPI version (*PGIMPI**)
- Re-implement linked-list conversion, which is really simple currently

Further Investigation/Mitigation

- MPI version shipped with PGI on POWER is slow, because it issues many instructions
- Further study: Identical assembly code generated as MPI-less version…
- … but includes call to `malloc()`!
- Different libraries linked for PGI and PGIMPI cases!
- `LD_PRELOAD=/lib64/libc.so.6` solves problem!

⇒ Slow MPI-aware `malloc()`?

- Mitigation
  - **Bug reported**
  - For now: consider as *anomalous* overhead

JÜLICH
FORSCHUNGSZENTRUM

Performance Modelling

Effective Bandwidth

Defining the model

JÜLICH FORSCHUNGSZENTRUM

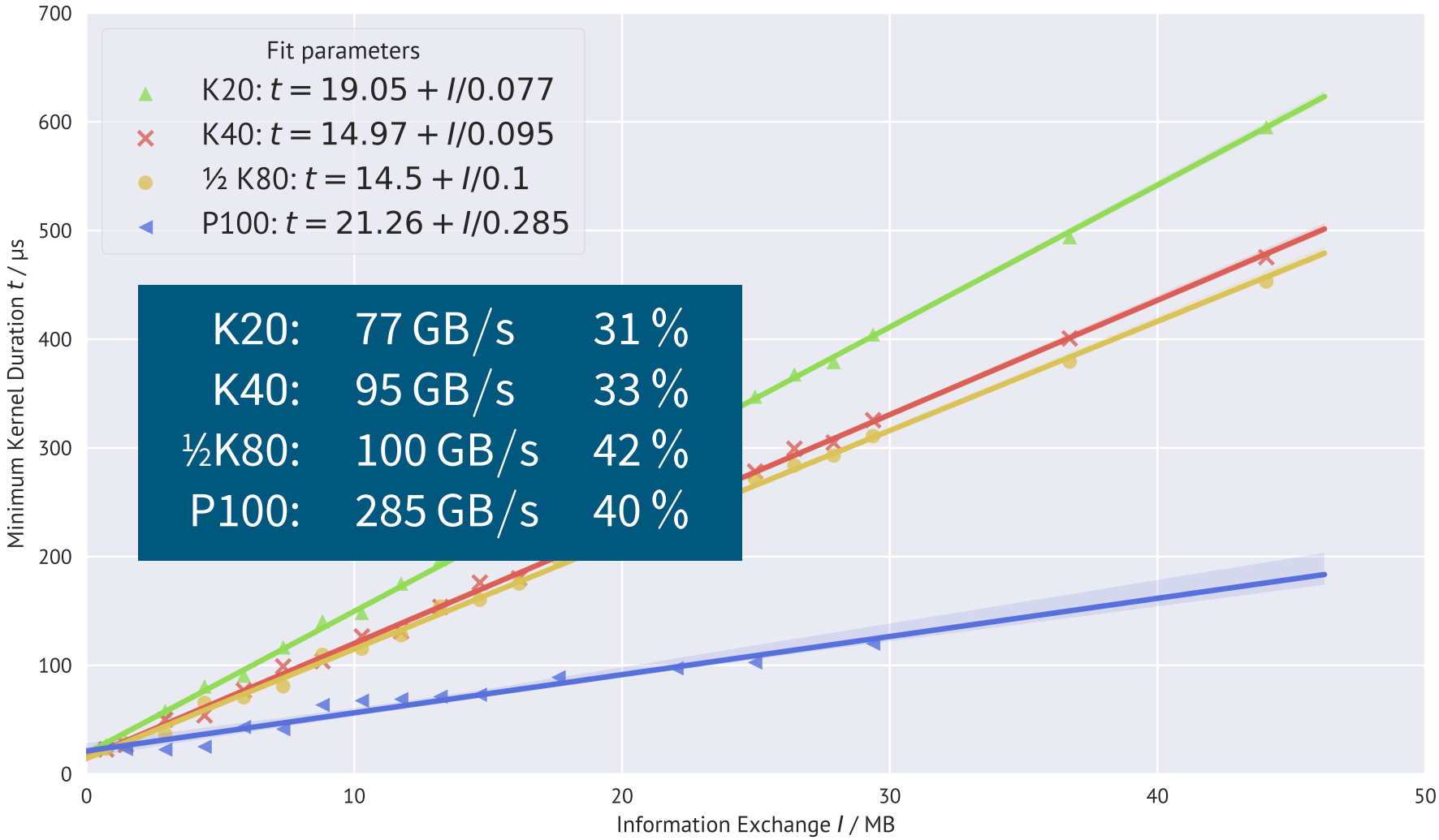
- **Goal**: Compare different GPU architectures; understand behavior of JuSPIC
- Model based on **information exchanged** of GPU kernel
  - Amount of exchanged information for given number of particles
  - Time for exchange

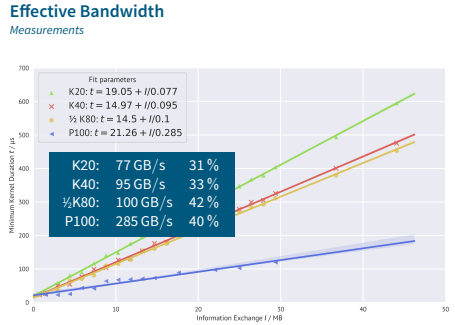
$$t(N_{\text{part}}) = \alpha + I(N_{\text{part}})/\beta\,,$$

N_{part}	Number of particles processed
I	Information exchanged (572 B (read) + 40 B (write))
t	Kernel runtime
α, β	Fit parameters; β: *effective bandwidth*

Effective Bandwidth

Measurements





- Used systems
  - JUHYDRA: K20, K40
  - JURECA: K80
  - JURON: P100
- GPU boost deactivated
- Relative numbers: utilized bandwidth (wrt theoretical max. bandwidth)
- JURON: Model relative to STREAM bandwidth: 50 %
- Pascal architecture beneficial
  - HBM2 memory $\rightarrow$ more throughput
  - More multiprocessors $\rightarrow$ more computations per time; more threads in flight
- Limit: Latency (kernel uses many registers)

Clock Dependency

Defining the relation

JÜLICH FORSCHUNGSZENTRUM

- Another free parameter: **GPU clock rates**
- Varies significantly across GPU architecture generations and models

$\rightarrow$ Incorporate clock into performance model

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

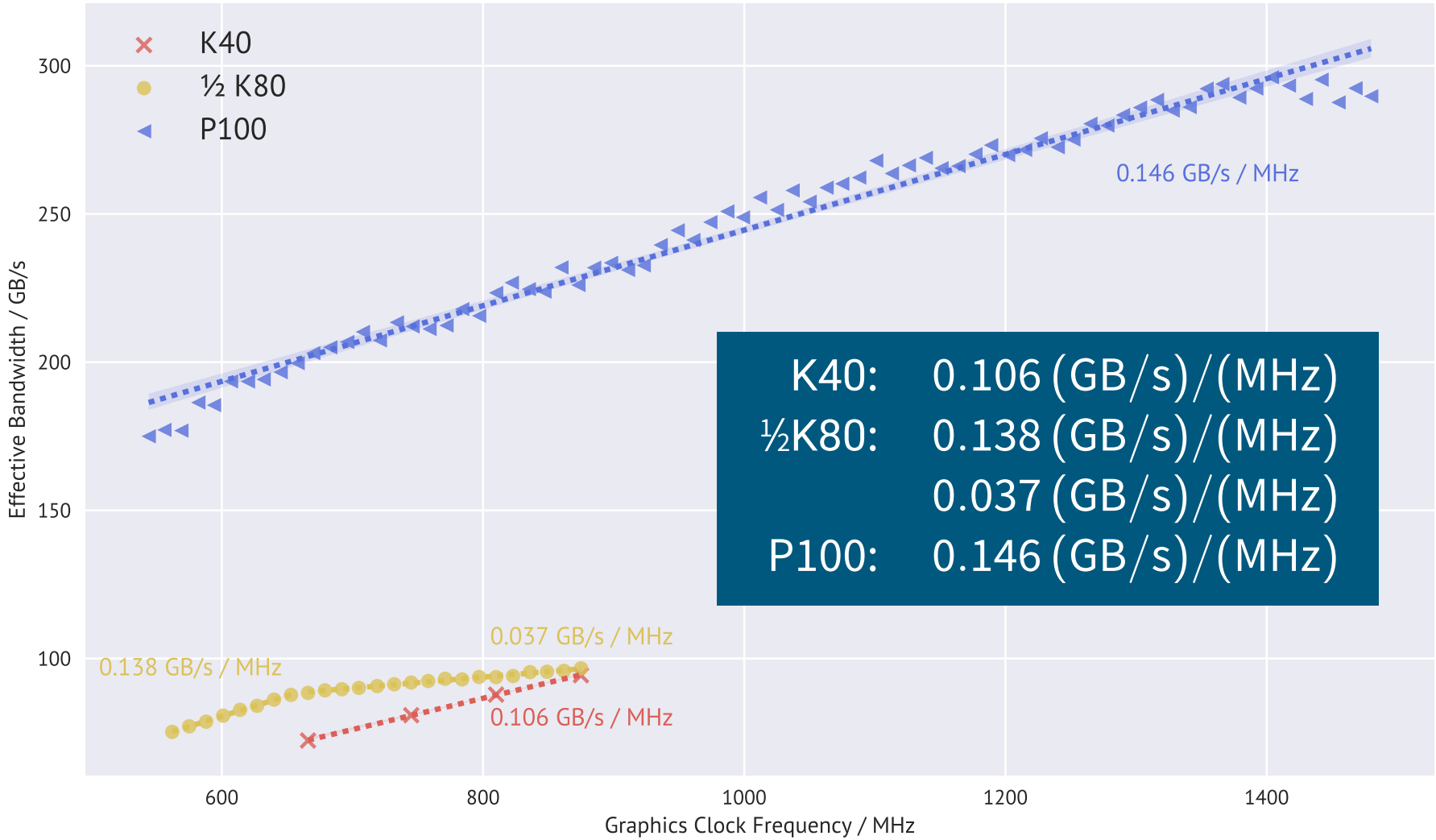
$\mathcal{C}$ GPU clock rate

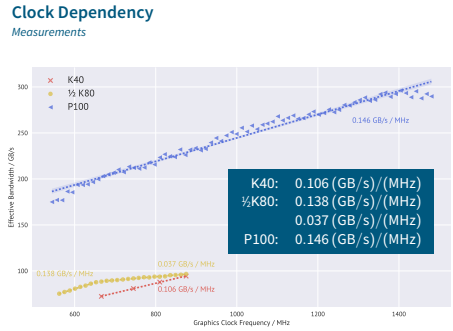
β Effective bandwidth (from before)

γ, δ Fit parameters

Clock Dependency

Measurements





- Strategy for measurement:
  1. Fix clock rate to value $\mathcal{C}_0$
  2. Determine effective bandwidth through model (β_0)
  3. Plot all $\mathcal{C}_i$ and β_i
  4. After all possible rates, linear-fit γ and δ
- Discussion
  - P100 very good for high clock rates, but also for rates of K40 and K80 (because memory bandwidth, #SM, …)
  - K80, K40 have similar chips; but K80 more efficient
  - Kink at K80: Here highest performance reached

Summary, Conclusion

JÜLICH FORSCHUNGSZENTRUM

Summary

- Enabled JuSPIC for **GPU** with OpenACC & CUDA Fortran
- Particle data layout: **SoA fastest**
- **Slow memory allocation** with PGI+MPI on POWER → bug filed
- Performance model: **Information exchange** (P100: 285 GB/s)
- Studied model with **different clock rates** – P100 most efficient scaling

Future

- Port also Reducer to GPU
- Enable MPI again
- Alternatives to linked list

Thank you for your attention!

a.herten@fz-juelich.de

JÜLICH
FORSCHUNGSZENTRUM

Appendix

Acknowledgements

- The work was done in context of two groups:

  POWER Acceleration and Design Centre A collaboration of IBM, NVIDIA, and Forschungszentrum Jülich

  NVIDIA Application Lab A collaboration of NVIDIA and Forschungszentrum Jülich

- Many thanks to Jiri Kraus from NVIDIA, who helped tremendously along the way

- JURON, a prototype system for the Human Brain Project, received co-funding from the European Union (Grant Agreement No. 604102)

Related Work

- Selection of other GPU PiC codes

  PSC The code JuSPIC is based on has been reimplemented in C and ported to GPU [4]

  PIConGPU PiC code specifically developed for GPUs [5]

- Minsky porting experiences
  - "Addressing Materials Science Challenges Using GPU-accelerated POWER8 Nodes" [6]
  - "A Performance Model for GPU-Accelerated FDTD Applications" [7]
- … more in paper!

OpenACC Performance Progression

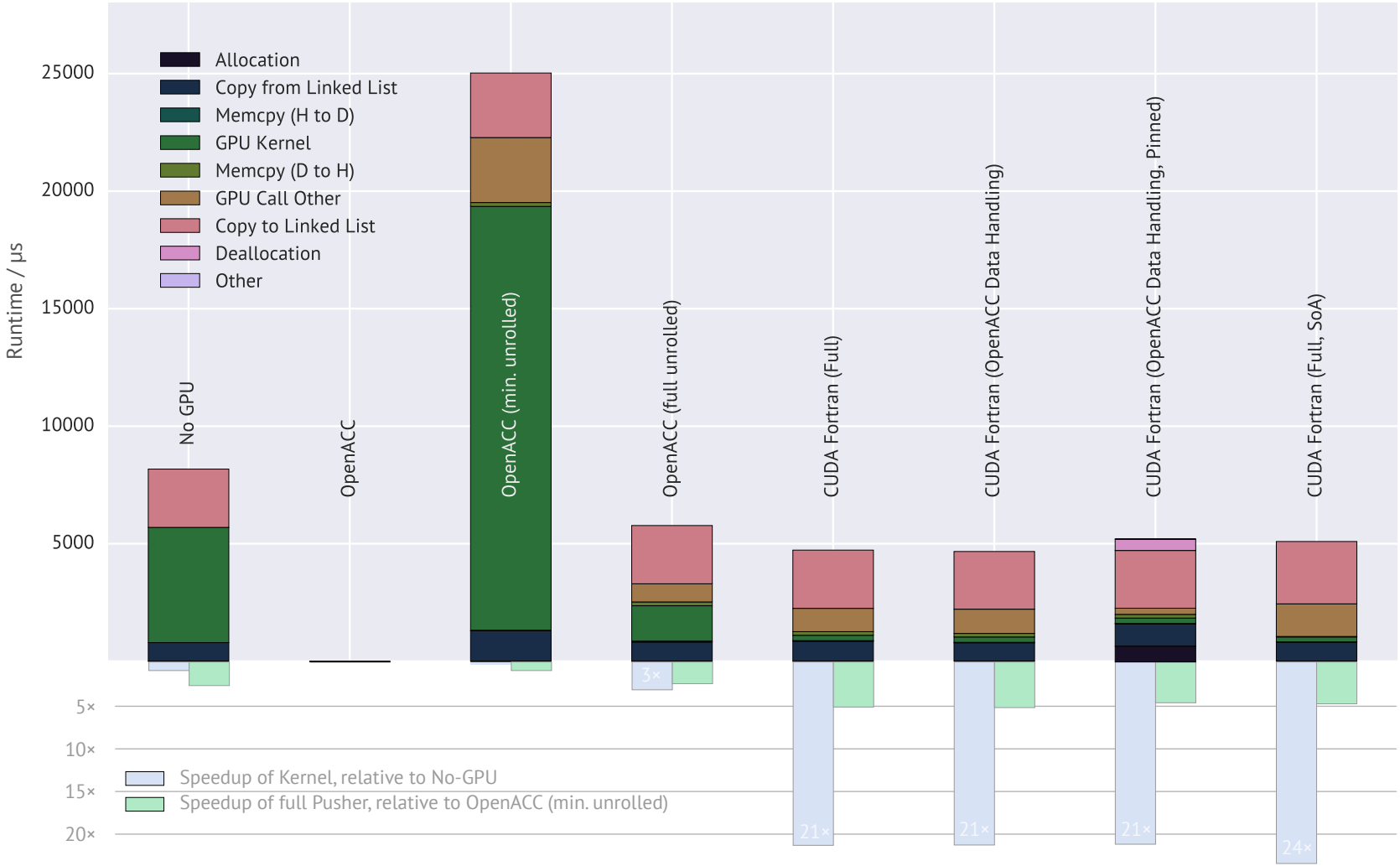
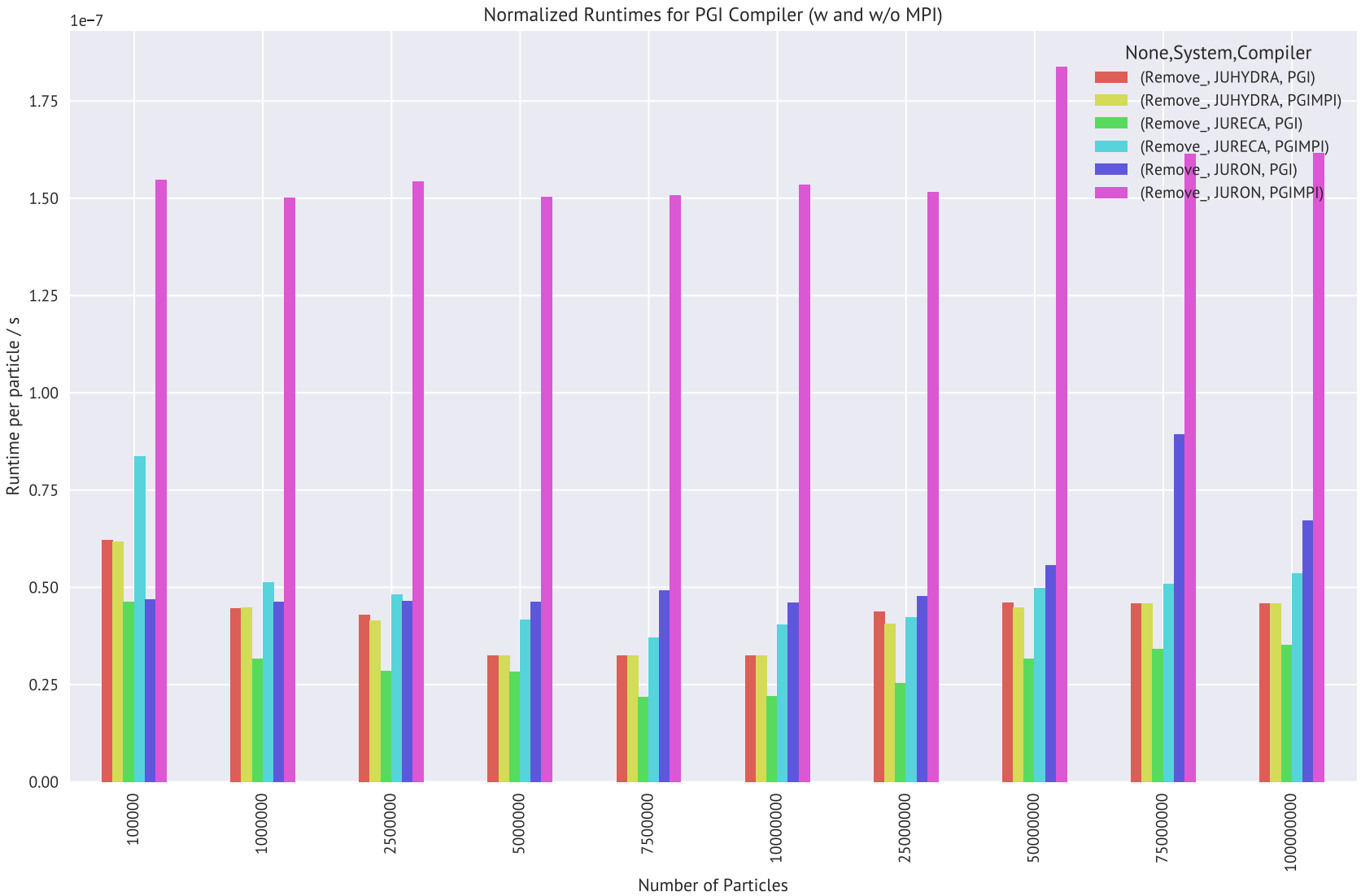


Figure: See GTC poster for details [8].

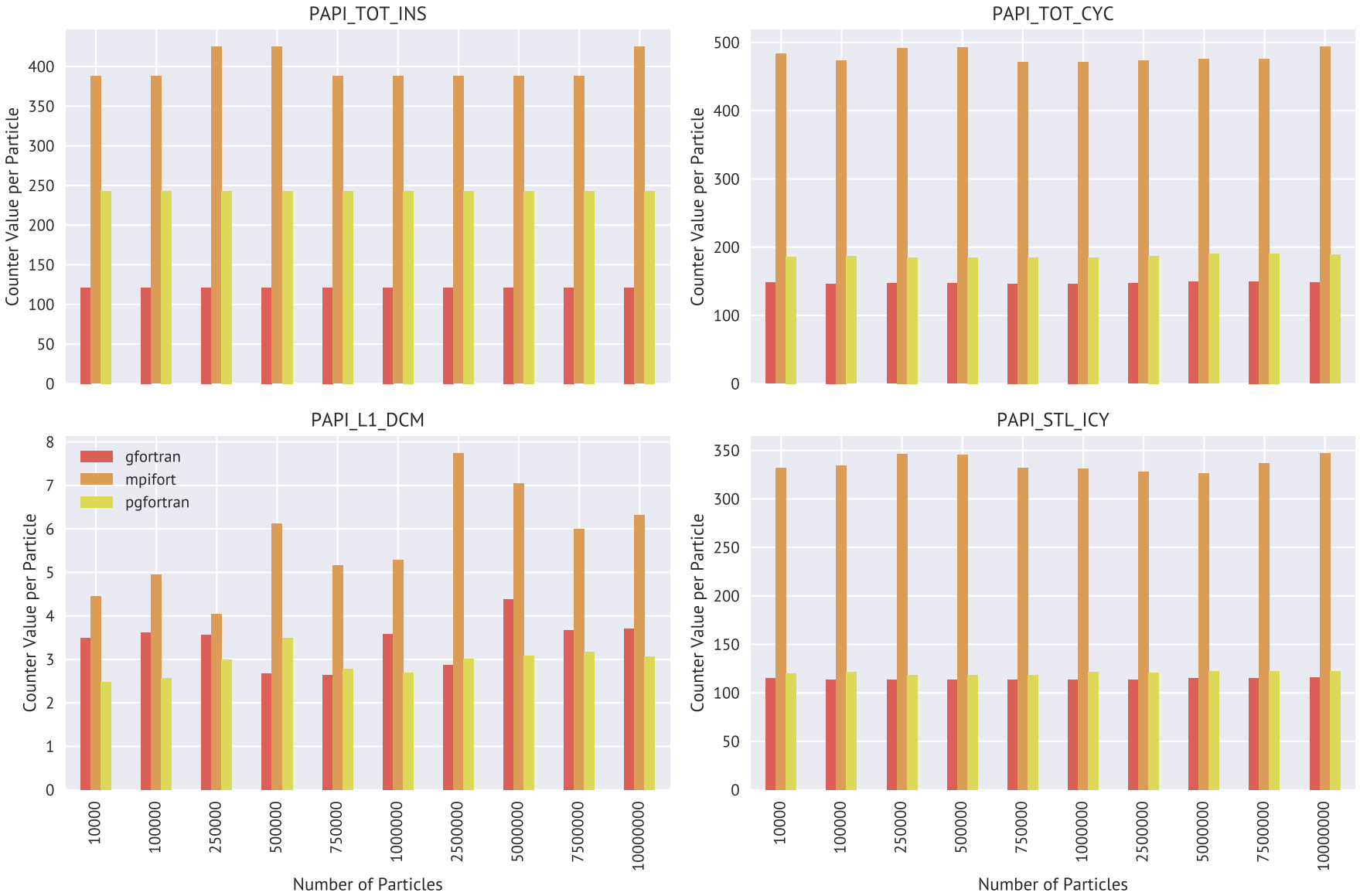
Linked List: Time for Remove on JURON

For different compilers



Selected Performance Counters on JURON

For different compilers



References: Images, Graphics I

[1] Forschungszentrum Jülich. *Hightech made in 1960: A view into the control room of DIDO*. URL: http://historie.fz-juelich.de/60jahre/DE/Geschichte/1956-1960/Dekade/_node.html (page 3).

[2] Forschungszentrum Jülich. *Forschungszentrum Bird's Eye*. (Page 3).

References I

[3] Phil Mucci and The ICL Team. *PAPI, the Performance Application Programming Interface*. URL: http://icl.utk.edu/papi/ (visited on 04/30/2017) (pages 35, 59).

[4] K. Germaschewski et al. "The Plasma Simulation Code: A modern particle-in-cell code with load-balancing and GPU support". In: *ArXiv e-prints* (Oct. 2013). arXiv: 1310.7866 [physics.plasm-ph] (page 49).

[5] M. Bussmann et al. "Radiative signature of the relativistic Kelvin-Helmholtz Instability". In: *2013 SC - International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. Nov. 2013, pp. 1–12. DOI: 10.1145/2503210.2504564 (page 49).

References II

[6] Paul F. Baumeister et al. “Addressing Materials Science Challenges Using GPU-accelerated POWER8 Nodes”. In: *Euro-Par 2016: Parallel Processing: 22nd International Conference on Parallel and Distributed Computing, Grenoble, France, August 24-26, 2016, Proceedings*. Ed. by Pierre-François Dutot and Denis Trystram. Cham: Springer International Publishing, 2016, pp. 77–89. ISBN: 978-3-319-43659-3. DOI: 10.1007/978-3-319-43659-3_6. URL: http://dx.doi.org/10.1007/978-3-319-43659-3_6 (page 49).

References III

[7] P. F. Baumeister et al. “A Performance Model for GPU-Accelerated FDTD Applications”. In: *2015 IEEE 22nd International Conference on High Performance Computing (HiPC)*. Dec. 2015, pp. 185–193. DOI: 10.1109/HiPC.2015.24 (page 49).

[8] Andreas Herten, Dirk Pleiter, and Dirk Brömmel. *Accelerating Plasma Physics with GPUs (Poster)*. Tech. rep. GPU Technology Conference, 2017 (page 50).

[9] Philip J. Mucci et al. “PAPI: A Portable Interface to Hardware Performance Counters”. In: *In Proceedings of the Department of Defense HPCMP Users Group Conference*. 1999, pp. 7–10 (page 59).

Glossary I

JÜLICH FORSCHUNGSZENTRUM

CUDA Computing platform for GPUs from NVIDIA. Provides, among others, CUDA C/C++. 2, 19, 21, 22, 46

FZJ Forschungszentrum Jülich, a research center in the west of Germany. 3, 57

JSC Jülich Supercomputing Centre operates a number of large and small supercomputers and connected infrastructure at FZJ. 3

JuSPIC Jülich Scalable Particle-in-Cell Code. 2, 6, 7, 8, 9, 12, 14, 18, 21, 35, 36, 40, 46, 49

Glossary II

JÜLICH FORSCHUNGSZENTRUM

MPI The Message Passing Interface, a communication message-passing application programmer interface. 35, 37, 38, 46

NVIDIA US technology company creating GPUs. 3, 5, 48, 57

NVLink NVIDIA's communication protocol connecting CPU $\leftrightarrow$ GPU and GPU $\leftrightarrow$ GPU with 80 GB/s. PCI-Express: 16 GB/s. 5, 57

OpenACC Directive-based programming, primarily for many-core machines. 2, 13, 14, 17, 18, 19, 22, 23, 26, 28, 46, 47, 50

P100 A large GPU with the Pascal architecture from NVIDIA. It employs NVLink as its interconnect and has fast *HBM2* memory. 2, 5, 30, 41, 44, 45, 46

Glossary III

JÜLICH FORSCHUNGSZENTRUM

PAPI The Performance API, a interface for accessing performance counters, also with aliased names cross-platform [3, 9]. 35, 36

Pascal The latest available GPU architecture from NVIDIA. 57

PGI Formerly *The Portland Group, Inc.*; since 2013 part of NVIDIA. 2, 21, 37, 46

PiC Particle in Cell; a method applied in a group of (plasma) physics simulations to solve partial differential equations. 2, 8, 49

POWER Series of microprocessors from IBM. 2, 3, 37, 46, 48

Tesla The GPU product line for general purpose computing computing of NVIDIA. 5