JÜLICH
FORSCHUNGSZENTRUM

Accelerating Plasma Physics with GPUs

PADC Annual Workshop 2016

Andreas Herten, Forschungszentrum Jülich, 17 October 2016

Contents

- JuSPIC on GPU
- With OpenACC
- Not with OpenACC
- Performance Model

JuSPIC

- Introduction
- Stages of Program

Acceleration

- Overview
- OpenACC
- CUDA Fortran
- CUDA OpenACC Hybrid
- SoA
- Speed-Up

Performance Model

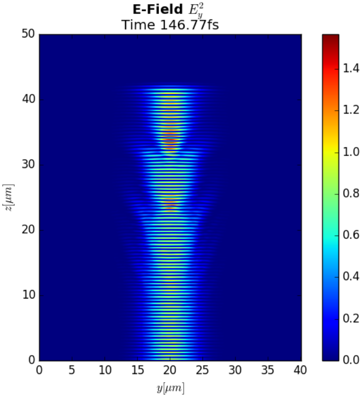
- Introduction
- Bandwidth
- Clock Frequency
- Normalization

Conclusion

JuSPIC: Introduction

man juspic

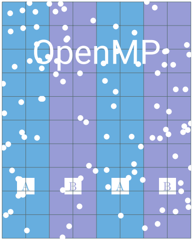
- **JuSPIC**: Jülich Scalable Particle-in-Cell Code
- Based on PSC by H. Ruhl
- Developed at JSC by SimLab Plasma Physics
- 3D electromagnetic Particle-in-Cell
- Properties
  - Solves relat. Vlasov equations, Maxwell equations
  - Scheme: Finite-difference time-domain
  - Cartesian geometry
  - Arbitrary number of particle species

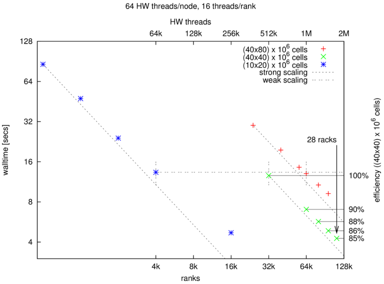


JuSPIC: Technologies

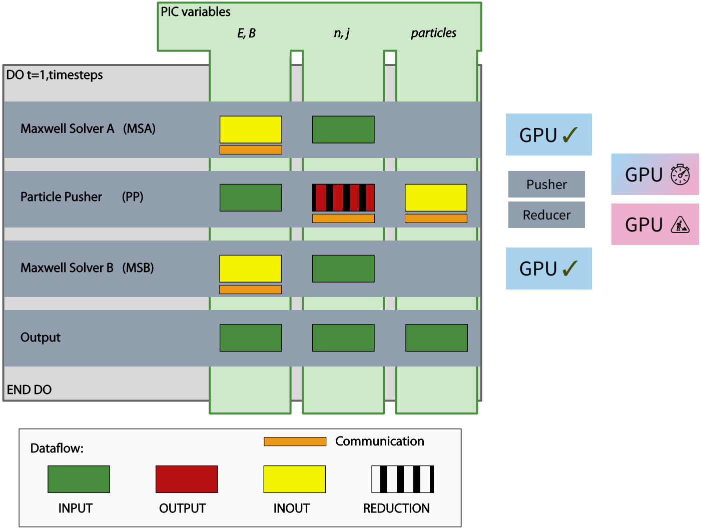
A quite parallel code

- Modern Fortran
- Distributed with **MPI**
  Domain decomposition: 3D
- CPU-parallelized with **OpenMP**
  Domain decomposition: Slices
- Particles connected by linked list
- High-Q Club:
  Scales to full JUQUEEN





Stages of Program



Acceleration

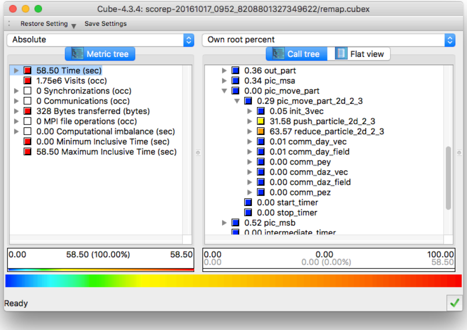
Accelerating JuSPIC

Start of the journey

- *Initial* requirements
  - — Leverage parallelism offered by GPUs
  - — Still work on all platforms
  - — Optimizations not solely for GPUs
  - → **OpenACC**
- Profiling
  1. Particle Reducer (64 %)
  2. Particle Pusher (32 %)
  3. **Maxwell Solver (1 %)**

OpenACC:

- Many cores with `pragmas`
- *(Not)* like OpenMP
- NVIDIA, AMD, …
- PGI, Cray, GCC
- C, Fortran



Cube-4.3.4: scorep-20161017_0952_8208801327349622/remap.cubex

Absolute | Own root percent

```
58.50 Time (sec)
1.75e6 Visits (occ)
0 Synchronizations (occ)
0 Communications (occ)
328 Bytes transferred (bytes)
0 MPI file operations (occ)
0.00 Computational imbalance (sec)
0.00 Minimum Inclusive Time (sec)
58.50 Maximum Inclusive Time (sec)
```

```
0.36 out_part
0.34 pic_msa
0.00 pic_move_part
  0.29 pic_move_part_2d_2_3
    0.05 init_3vec
    31.58 push_particle_2d_2_3
    63.57 reduce_particle_2d_2_3
    0.01 comm_day_vec
    0.01 comm_day_field
    0.00 comm_pey
    0.00 comm_daz_vec
    0.00 comm_daz_field
    0.00 comm_pez
  0.00 start_timer
  0.00 stop_timer
0.52 pic_msb
0.00 intermediate_timer
```

```
0.00      58.50 (100.00%)      58.50
```

```
0.00      0.00                 100.00
0.00      0.00 (0.00%)         58.50
```

Maxwell Solver

Straight forward

- Content of function: Update $\vec{E}$ and $\vec{B}$ fields
- Global data handling
- Source example

```
!$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  +
        ↪   cny*(b(i1,i2,i3)%Z-b(i1,i2-1,i3)%Z) -
        ↪   cnz*(b(i1,i2,i3)%Y-b(i1,i2,i3-1)%Y) -
        ↪   0.5*dt*ji(i1,i2,i3)%X
        ! ...
```

Maxwell Solver

Straight forward

- Content of function: Update $\vec{E}$ and $\vec{B}$ fields
- Global data handling

```
advance_e_vol:
  1410, Generating present(e(:,:,:),b(:,:,:),ji(:,:,:))
  1412, Loop is parallelizable
  1414, Loop is parallelizable
  1415, Loop is parallelizable
        Accelerator kernel generated
        Generating Tesla code
     1412, !$acc loop gang, vector(128) collapse(3) ! blockidx%x threadidx%x
     1414,   ! blockidx%x threadidx%x collapsed
     1415,   ! blockidx%x threadidx%x collapsed
```

```
↪   ch2*(b(i1,i2,i3)%Y - b(i1,i2,i3-1)%Y)
↪   0.5*dt*ji(i1,i2,i3)%X
! ...
```

✓

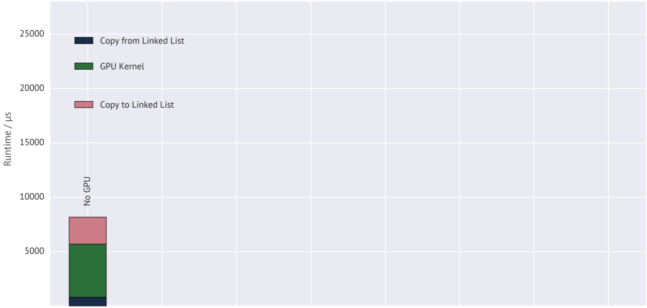
Particle Pusher

Start of a journey…

- Content of function:
  Interpolate field, update particle position and momentum
- Change to source: Linked list of particles → array of particles
- Timings

  ***CPU**: Intel Xeon Sandy Bridge (2 GHz), no MPI, no OpenMP*

  ***GPU**: NVIDIA Tesla K40, ECC enabled*



Acceleration with OpenACC

This should be easy, right?

- Simple addition in front of code

```
!$acc parallel loop private(pp,root,qi,mi,wi) present(e,
↪  b) copy(list_of_particles)
do i_particle = loop_min, loop_max
   x_(:)=list_of_particles(i_particle)%vec(:)
   p_(:)=list_of_particles(i_particle)%pvec(:)
   qi   =list_of_particles(i_particle)%q

   mi=p_prop(list_of_particles(i_particle)%id)%m
   wi=p_prop(list_of_particles(i_particle)%id)%w

   root=1.0/sqrt(1.0+sum(p_**2))
   !...
```

Acceleration with OpenACC

This should be easy, right?

- Simple addition in front of code

```
!$acc parallel loop private(pp,root,qi,mi,wi) present(e,
```

```
push_particle_2d_2_3:
  875, include 'pic.in.gpu.minimallyunrolled.F90'
       254, Generating present(e(:,:,:),b(:,:,:))
            Generating copy(list_of_particles(:))
            Accelerator kernel generated
            Generating Tesla code
           255, !$acc loop gang, vector(128) ! blockidx%x threadidx%x
       254, Generating copyout(tvec3(:),tvec2(:),tvec1(:))
            Generating copyin(xyzl(2:),p_prop(list_of_particles%id))
            Generating copyout(x_(:),p_(:),v_(:))
            ...
```

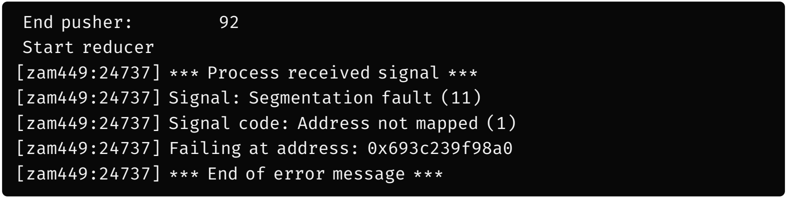
```
    root=1.0/sqrt(1.0+sum(p_**2))
    !...
```

Acceleration with OpenACC

This should be easy, right?

- Simple addition in front of code

```
!$acc parallel loop private(pp,root,qi,mi,wi) present(e,
↪  b) copy(list_of_particles)
```



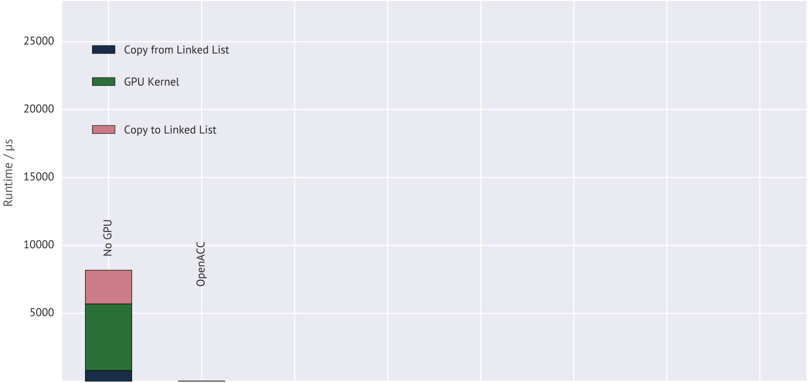
```
 End pusher:         92
 Start reducer
[zam449:24737] *** Process received signal ***
[zam449:24737] Signal: Segmentation fault (11)
[zam449:24737] Signal code: Address not mapped (1)
[zam449:24737] Failing at address: 0x693c239f98a0
[zam449:24737] *** End of error message ***
```

```
    wi=p_prop(list_of_particles(i_particle)%id)%w

    root=1.0/sqrt(1.0+sum(p_**2))
    !...
```

Acceleration with OpenACC

Well…



✗

OpenACC?

At least a working version?!

- Changes for a *running* PGI OpenACC program
  - Unroll some operations on arrays

    PGI compiler silently/automatically generates temporary variables which it stumbles over during OpenACC translation step

    ```
    x_(1)=list_of_particles(i_particle)%vec(1)
    x_(2)=list_of_particles(i_particle)%vec(2)
    x_(3)=list_of_particles(i_particle)%vec(3)
    p_(1)=list_of_particles(i_particle)%pvec(1)
    ! ...
    ```
  - Explicitly stage private variables

    ```
    !$acc loop private(bvp,x_,hh,jj,t,...
    ```
  - Limit number of threads!

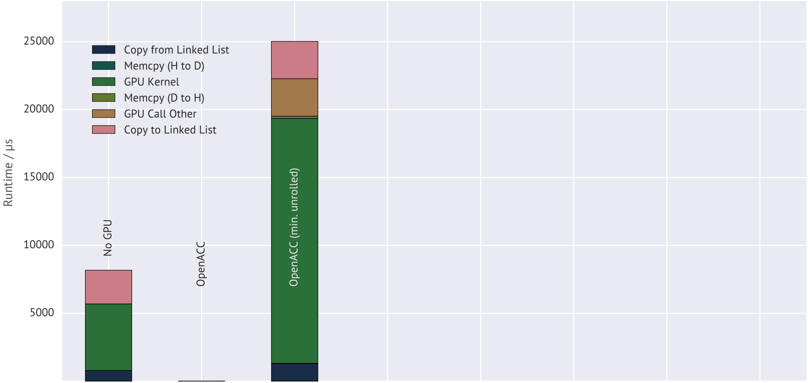
    Too much state?!

    ```
    !$acc num_gangs(2) vector_length(8)
    ```

→ ***Slow!?***

OpenACC?

Slow!



. . . ?!

OpenACC with Speedup

Finally!

- Changes for a *fast* PGI OpenACC program
  - Replace all arrays with scalars, all operations on arrays with scalar operations

    Preprocessor macros to the rescue!

    ```
    xi_1 = list_of_particles(i_particle)%vec(1)
    xi_2 = list_of_particles(i_particle)%vec(2)
    xi_3 = list_of_particles(i_particle)%vec(3)
    pi_1 = list_of_particles(i_particle)%pvec(1)
    ! ...
    ```
  - No limiting of threads, straight-forward `!$acc` statement
- Not much Fortran left

OpenACC with Speedup

Finally!

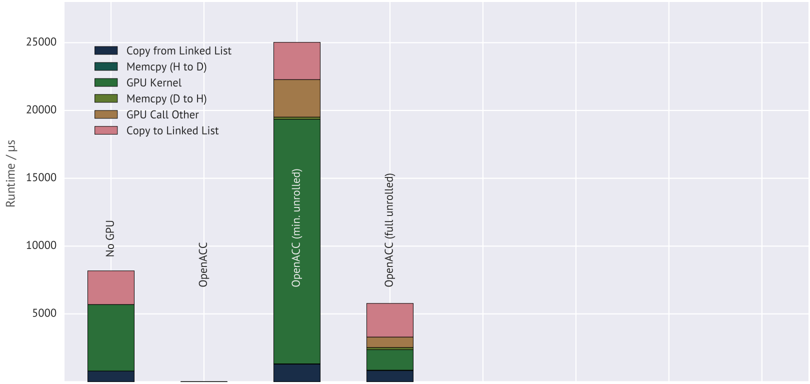
- Changes for a *fast* PGI OpenACC program
  - Replace all arrays with scalars, all operations on arrays with scalar operations

```
push_particle_2d_2_3:
   875, include 'pic.in.gpu.fullyunrolled.F90'
       268, Generating present(e(:,:,:),b(:,:,:))
           Generating copy(list_of_particles(:))
           Accelerator kernel generated
           Generating Tesla code
          269, !$acc loop gang, vector(128) ! blockidx%x threadidx%x
       268, Generating copyin(p_prop(list_of_particles%id))
```

  - No limiting of threads, straight-forward *!$acc* statement
- Not much Fortran left

OpenACC!

It's faster!



✓

CUDA Fortran

Let’s bring out the big guns

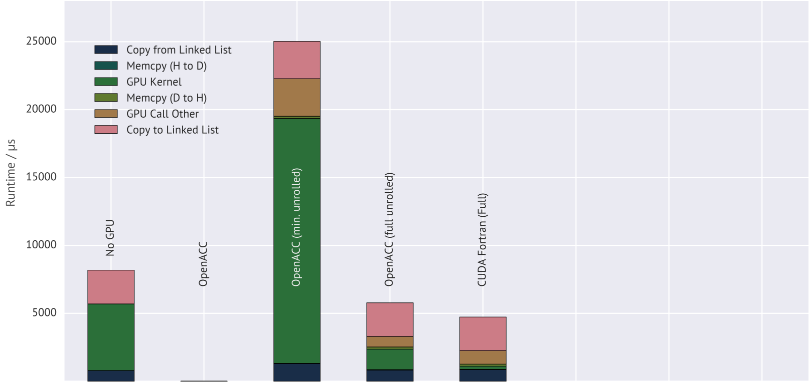
- At this point, code closer to rewritten C code than to original code
- Not very OpenACC-ish
- Different approach: **CUDA Fortran**!
- Can also be *portable* with pre-processor guards

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

- Original code can be kept!

CUDA Fortran!

A good time!



✓

Hybrid CUDA OpenACC

Mingling

- Already in last version:

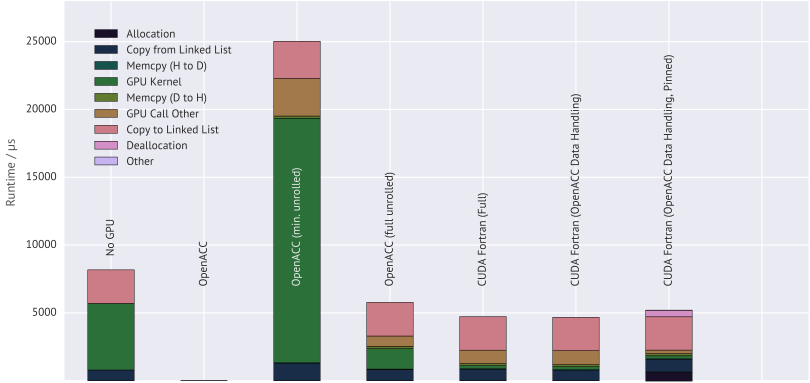
OpenACC	Maxwell Solver, *helper* data (scalars, 3D vectors, fields)
CUDA	Particle Pusher, particle momenta / positions

→ Evaluation of

- Full data handling with OpenACC
- Full data handling with OpenACC, pinned host memory

OpenACC data handling

OpenACC copy is reasonable



✓/✗

Data Structure of Particles

AoS → SoA

- Original data structure: Array of structs (AoS)

```
type particle
   sequence
   real(dp_kind) :: vec(3), pvec(3)
end type particle
type(particle), dimension(:), allocatable ::
↪  list_of_particles
```

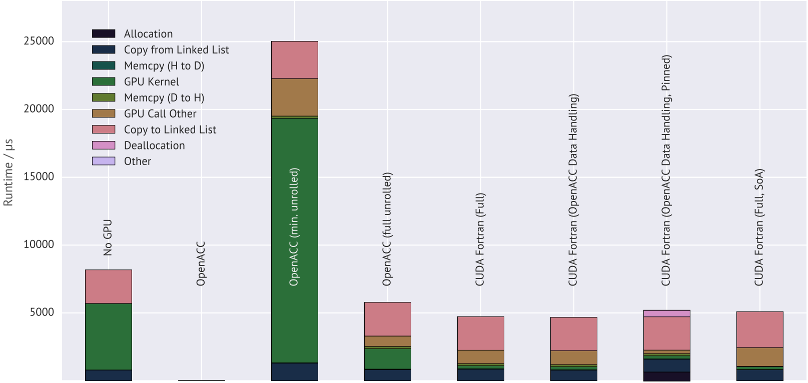
- Align data for coalesced GPU access (SoA)

```
type posmom
  real(dp_kind), dimension(:), allocatable :: x, y, z, px,
  ↪  py, pz
end type posmom
type(posmom) :: soa_list_of_particles
```

- Data only re-allocated when size changes

SoA Data Layout

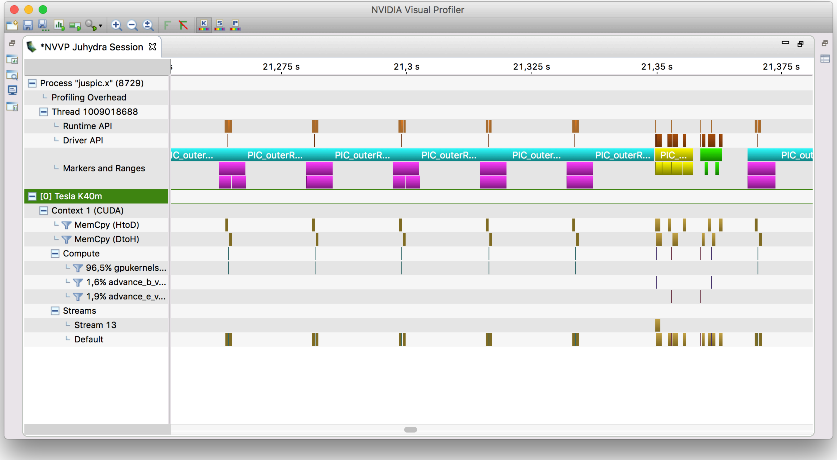
Worth only if data is touched anyway



✓

Visual Profiler

CUDA Fortran (Full, SoA)

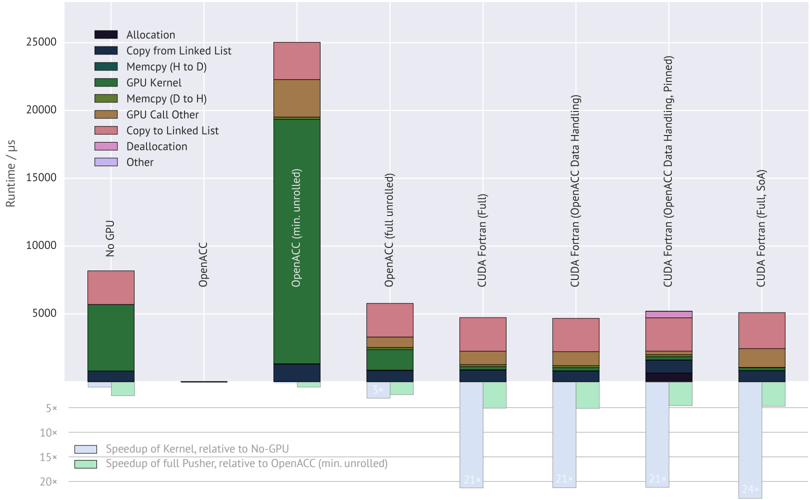


NVIDIA Visual Profiler

*NVVP Juhydra Session

Speed-Up

🟦 *Kernel to CPU;* 🟩 *Full pusher to OpenACC*



Performance Model

Introduction

Some information

- Simple **information exchange model**

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

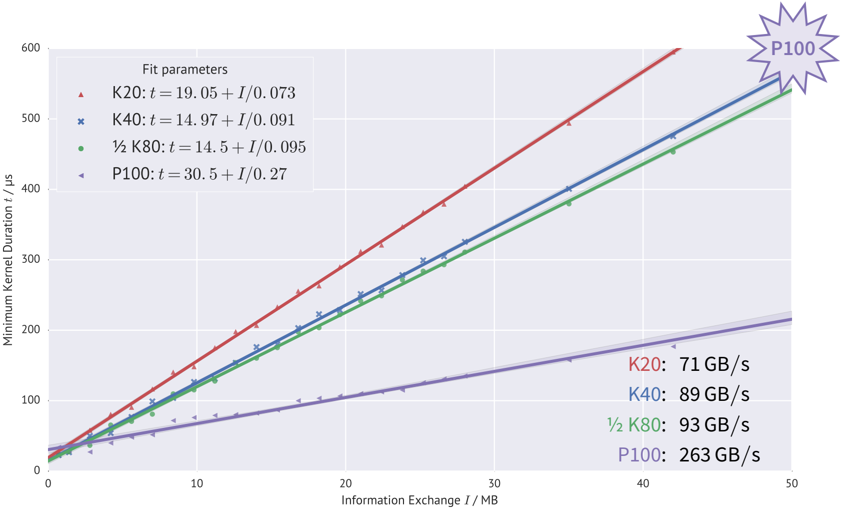
N_{part}	Number of particles processed
t	Duration of execution (in s)
I	Amount of information exchanged (in B)
α	Offset (*zero-data latency*); fit parameter
β	Slope (*effective bandwidth*); fit parameter

- Hypothesis: JuSPIC's GPU performance is largely limited by available bandwidth

→ β is lower limit of exploited bandwidth

Bandwidth Determination

GPU clock fixed to maximum value



K20: 71 GB/s

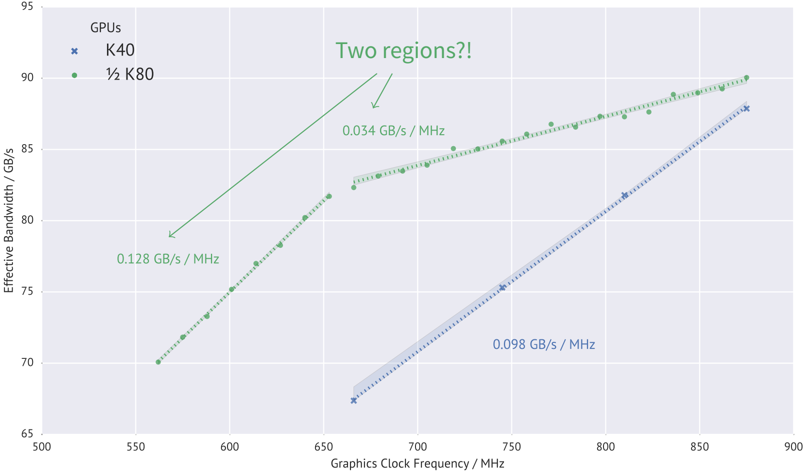
K40: 89 GB/s

½ K80: 93 GB/s

P100: 263 GB/s

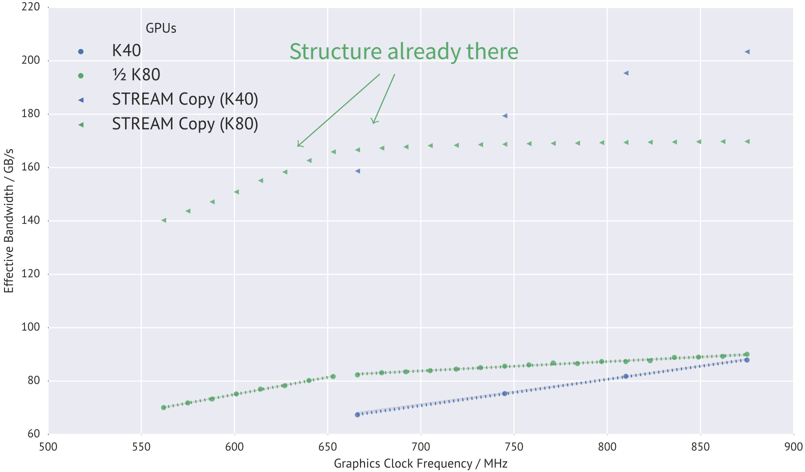
Bandwidth vs. Clock Frequency

Graphics clock frequency



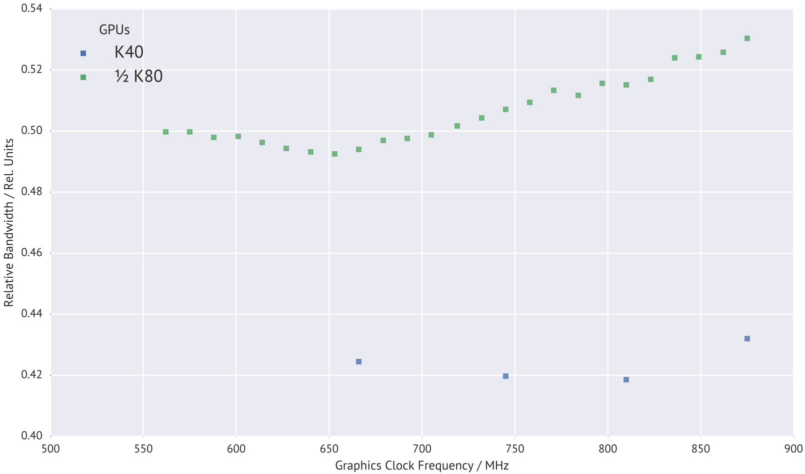
STREAM Bandwidth

As a means of normalization



Normalized Clock-dependent Bandwidth

Bandwidth vs. Clock Frequency, normalized to STREAM results



Results & Conclusions

- Performance Model
  - — Information exchange model: JuSPIC not bandwidth-limited
  - → Further investigation needed (Computation? **Latency?**)
  - — Peculiar: steps in STREAM (K80); *valley of efficiency* (K80, K40)
  - — More byte per clock cycle for ½K80 (before step)
- Porting with OpenACC
  - — JuSPIC's Fortran too complicated for OpenACC (7 bugs with PGI …)
  - — CUDA Fortran also portable, closer to original code
  - — Mixing OpenACC and CUDA Fortran feasible
  - — ½ of computing-heavy functions ported; promising
  - ⇒ Full effect only if H↔D copies reduced —

Thank you
for your attention!
a.herten@fz-juelich.de