Accelerating an Edge-Based CFD Solver Using Many-Core Co-Processors

Matthias Möller
Institute of Applied Mathematics (LS3)
TU Dortmund, Germany

Thanks to D. Göddeke, M. Köster, D. Ribbrock (TUDo)
D. Kuzmin (University of Erlangen-Nuremberg)

CFD-Software: FeatFlow2

- Open-source finite element library with demo applications
- ~ 700.000 lines of Fortran 95 code + external libraries
- available online http://www.featflow.de/en/software/featflow2.html

CFD-Software: FeatFlow2

- Open-source finite element library with demo applications
- ~ 700.000 lines of Fortran 95 code + external libraries
- available online http://www.featflow.de/en/software/featflow2.html

1. Milestone: *OpenMP parallelization*
    - classical FE-like assembly of coefficient matrices
    - edge-based assembly of vectors and operators
    - other loops (flux limiter, MVmult, norms, ...)

CFD-Software: FeatFlow2

- Open-source finite element library with demo applications
- ~ 700.000 lines of Fortran 95 code + external libraries
- available online http://www.featflow.de/en/software/featflow2.html

1. Milestone: *OpenMP parallelization*
   - classical FE-like assembly of coefficient matrices
   - edge-based assembly of vectors and operators
   - other loops (flux limiter, MVmult, norms, ...)

2. Milestone: *CUDA port of time consuming parts*
   - minimally invasive integration of co-processor support
   - edge-based assembly of vectors and operators

CFD-Software: FeatFlow2

- Open-source finite element library with demo applications
- ~ 700.000 lines of Fortran 95 code + external libraries
- available online http://www.featflow.de/en/software/featflow2.html

1. Milestone: *OpenMP parallelization*
   - classical FE-like assembly of coefficient matrices
   - edge-based assembly of vectors and operators
   - other loops (flux limiter, MVmult, norms, ...)

Reuse of application code via meta-programming library (C++/Fortran)

2. Milestone: *CUDA port of time consuming parts*
   - minimally invasive integration of co-processor support
   - edge-based assembly of vectors and operators

Galerkin finite element schemes $\partial_t U + \nabla \cdot \mathbf{F}(U) = 0$

- Weak formulation

$$\int_\Omega W \frac{\partial U}{\partial t} - \nabla W \cdot \mathbf{F}(U)\,\mathrm{d}\mathbf{x} + \int_\Gamma W\,\mathbf{n} \cdot \mathbf{F}(U)\,\mathrm{d}s = 0, \quad \forall W \in \mathcal{W}$$

- Group representation [Fle83]

$$U(\mathbf{x}, t) \approx \sum\nolimits_j \varphi_j(\mathbf{x}) U_j(t) \qquad \mathbf{F}(U) \approx \sum\nolimits_j \varphi_j(\mathbf{x}) \mathbf{F}(U_j)$$

- Semi-discrete high-order scheme

$$\sum\nolimits_j m_{ij} \frac{\mathrm{d}U_j}{\mathrm{d}t} - \sum\nolimits_j \mathbf{c}_{ji} \cdot \mathbf{F}_j + \sum\nolimits_j \mathbf{s}_{ij} \cdot \mathbf{F}_j = 0$$

$$m_{ij} = \int_\Omega \varphi_i \varphi_j\,\mathrm{d}\mathbf{x} \qquad \mathbf{s}_{ij} = \int_\Gamma \varphi_i \varphi_j \mathbf{n}\,\mathrm{d}s \qquad \mathbf{c}_{ji} = \int_\Omega \nabla\varphi_i \varphi_j\,\mathrm{d}\mathbf{x}$$

Galerkin finite element schemes, cont'd

- Galerkin flux decomposition

$$\sum_j \mathbf{c}_{ij} = 0 \quad \Rightarrow \quad -\sum_j \mathbf{c}_{ji} \cdot \mathbf{F}_j = \sum_{j \neq i} \boxed{\mathbf{c}_{ij} \cdot \mathbf{F}_i - \mathbf{c}_{ji} \cdot \mathbf{F}_j}$$

- Semi-discrete high-order scheme [Ku03]

$$\sum_j \left[m_{ij} \frac{\mathrm{d}U_j}{\mathrm{d}t} + \mathbf{s}_{ij} \cdot \mathbf{F}_j \right] + \sum_{j \neq i} G_{ij} = 0$$

  - efficient edge-based assembly of Galerkin fluxes
  - precomputation of coefficient matrices (on CPU) and singular transfer to device memory (low storage requirement on GPU)

Algebraic flux correction, *Kuzmin et al.*

- Semi-discrete low-order scheme

$$\underline{m_i \frac{\mathrm{d}U_i}{\mathrm{d}t}} + \sum_{j \neq i} G_{ij} + \underline{D_{ij}(U_j - U_i)} = 0 \qquad m_i = \sum\nolimits_j m_{ij}$$

mass lumping   artificial dissipation

- Conservative flux decomposition

$$m_i(U_i^H - U_i^L) = \sum_{j \neq i} \underline{m_{ij} \left(\frac{\mathrm{d}U_i}{\mathrm{d}t} - \frac{\mathrm{d}U_j}{\mathrm{d}t} \right) + D_{ij}(U_i - U_j)}$$

antidiffusive fluxes

- **low-order scheme + limited antidiffusion = high-resolution scheme**
- Parallelization of edge-loops is crucial to achive high overall efficiency

Outline of solution algorithm

- **Initialization:** Transfer edge-data to global device memory

In every **time step**:

- Transfer solution vector into global device memory
- Assemble rhs vector and transfer back to host memory

  In every **nonlinear step**:

  - Assemble nonlinear parts of operator and residual vector and transfer to host memory
  - Combine with constant contributions on host
  - Solve nonlinear problem, update solution, and transfer solution into global device memory

overlap transfers
with computation

Preparation of parallel edge-based assembly

- Edge-coloring of the FE sparsity graph

$c \leq 2\Delta - 1$	greedy algorithm
$\Delta \leq c \leq \Delta + 1$	Vizing's algorithm

- Precompute constant coefficient matrices (classical FE-assembly on CPU) and store them into edge-based data structure (AoS/SoA/mixture)

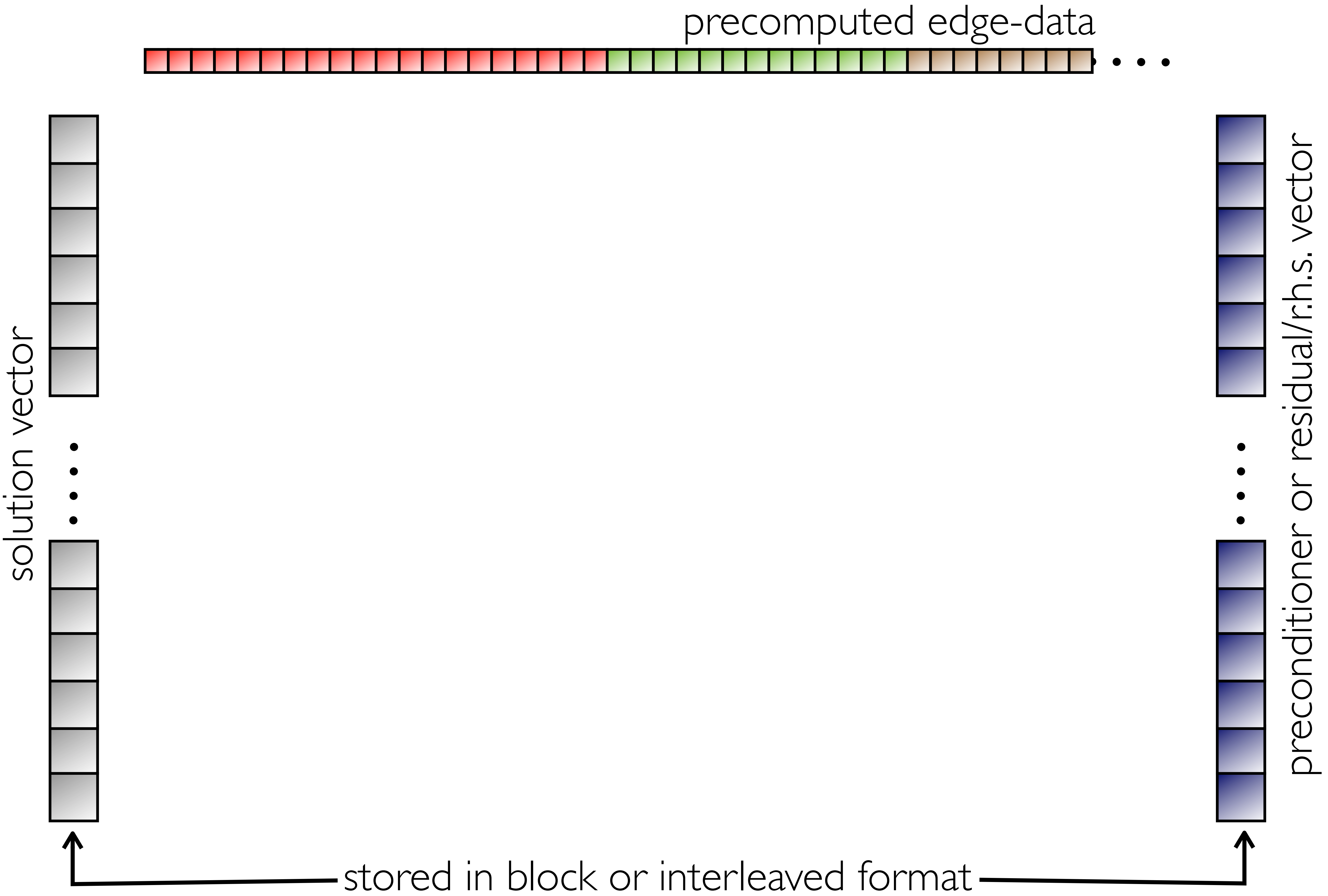
```
struct {
  int i, j;
  double m_ij;
  double cx_ij, cx_ji;
  double cy_ij, cy_ji;
} EdgeData[nedge];
```

or

```
int nodes[2][nedge];
double coeff_m[nedge];
double coeff_cx[2][nedge];
double coeff_cy[2][nedge];
```

and index vector separating color groups

Parallel edge-based assembly



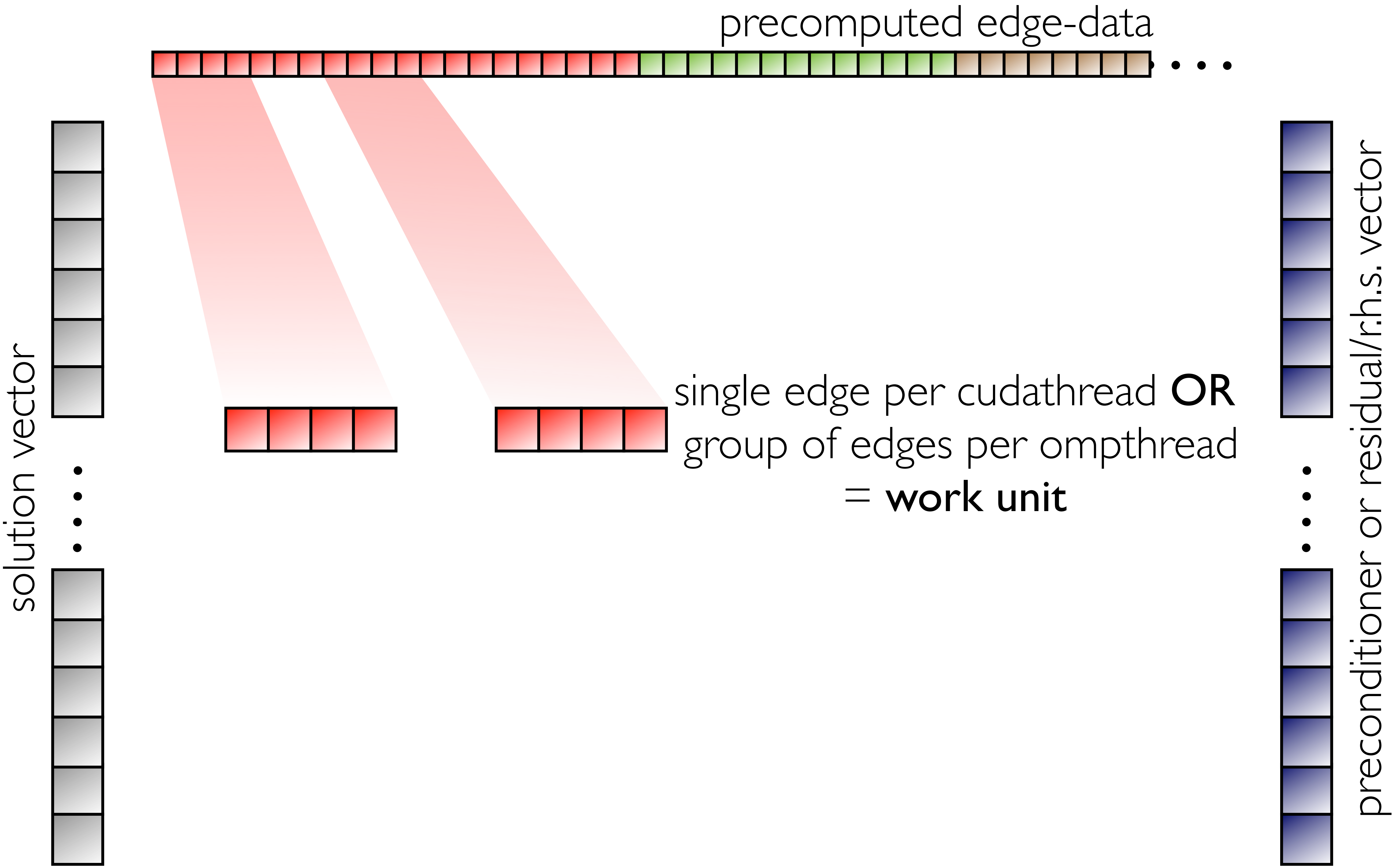
precomputed edge-data

solution vector

preconditioner or residual/r.h.s. vector

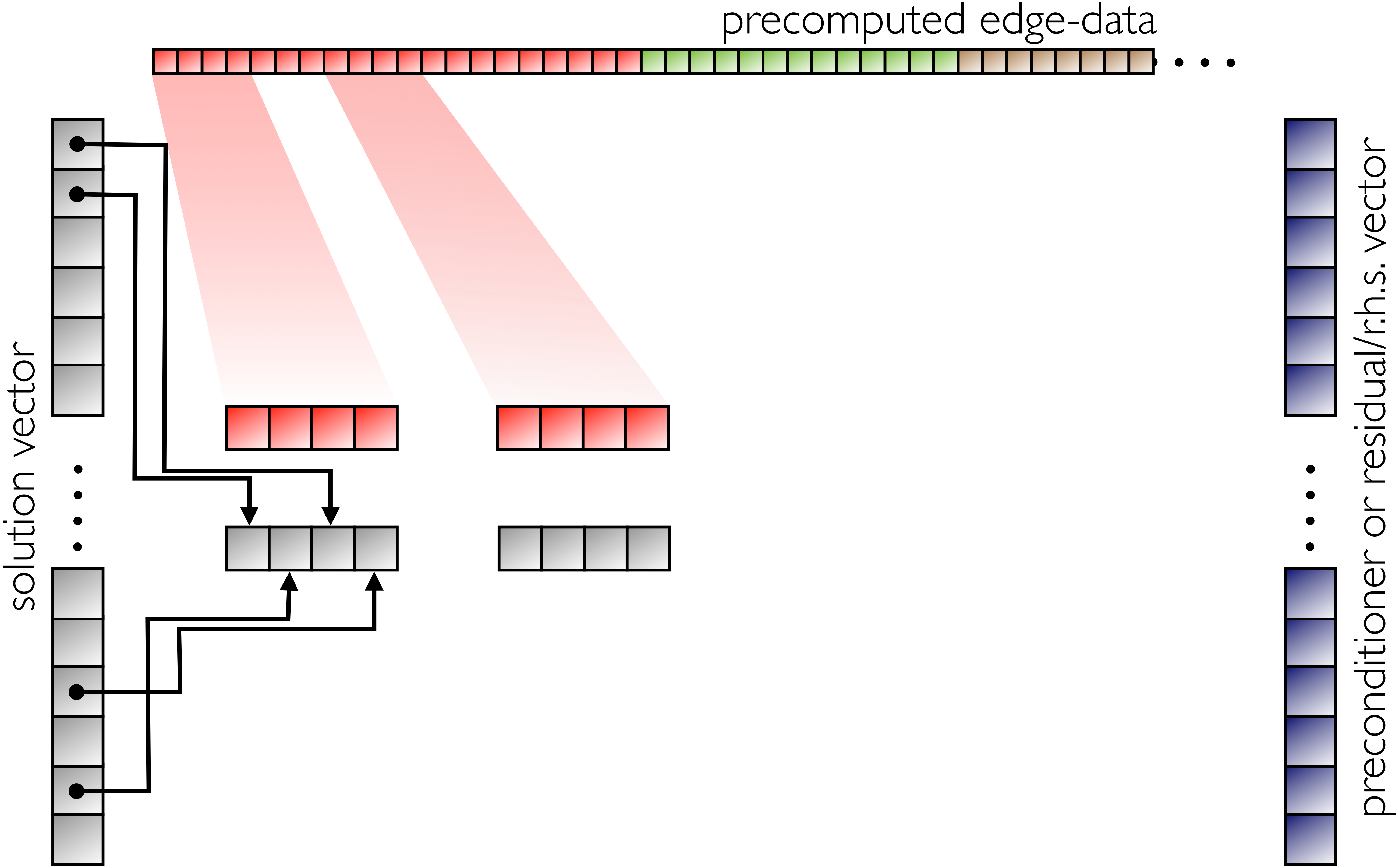
stored in block or interleaved format

Parallel edge-based assembly

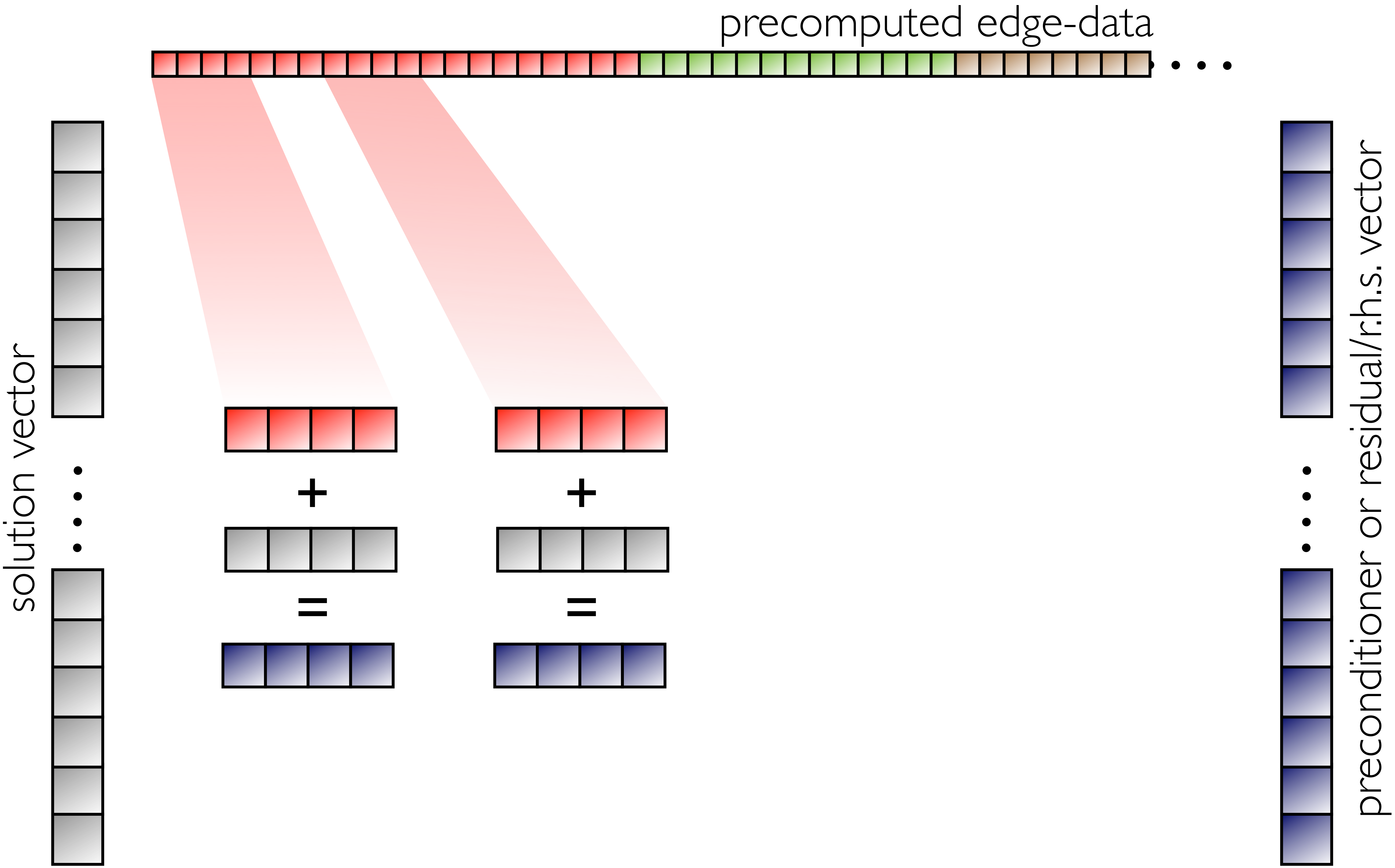


single edge per cudathread **OR**
group of edges per ompthread
= **work unit**

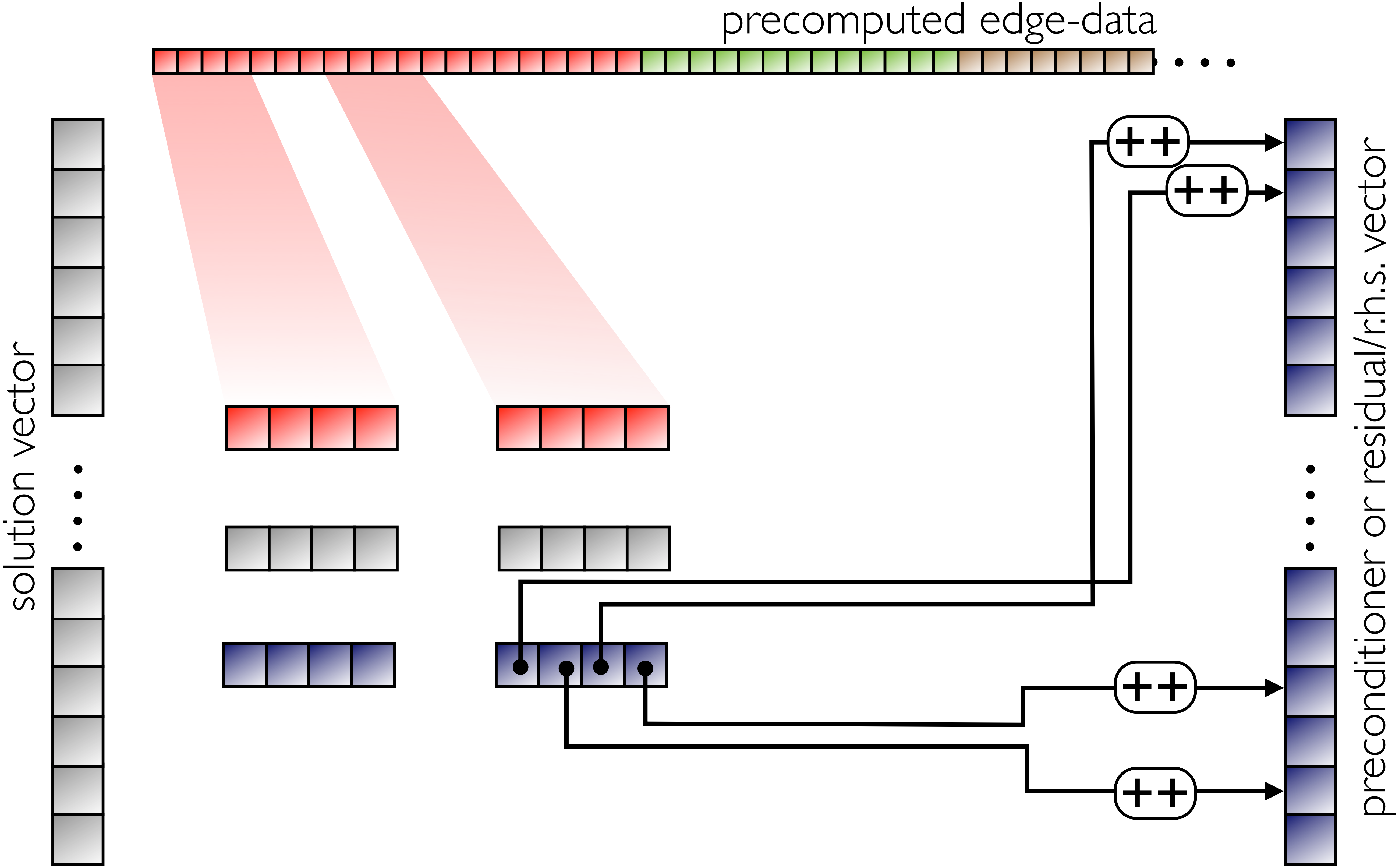
Parallel edge-based assembly



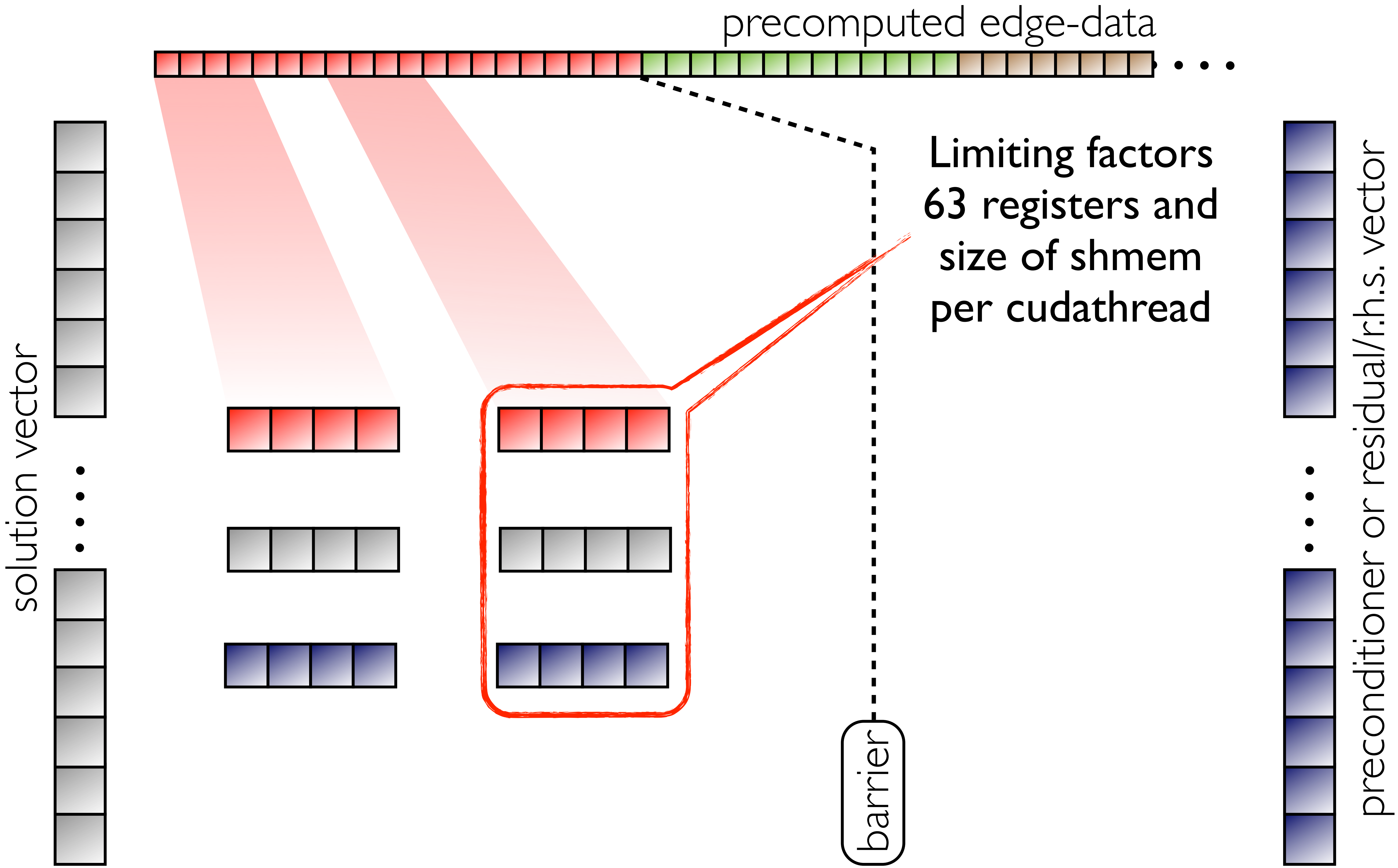
Parallel edge-based assembly



Parallel edge-based assembly



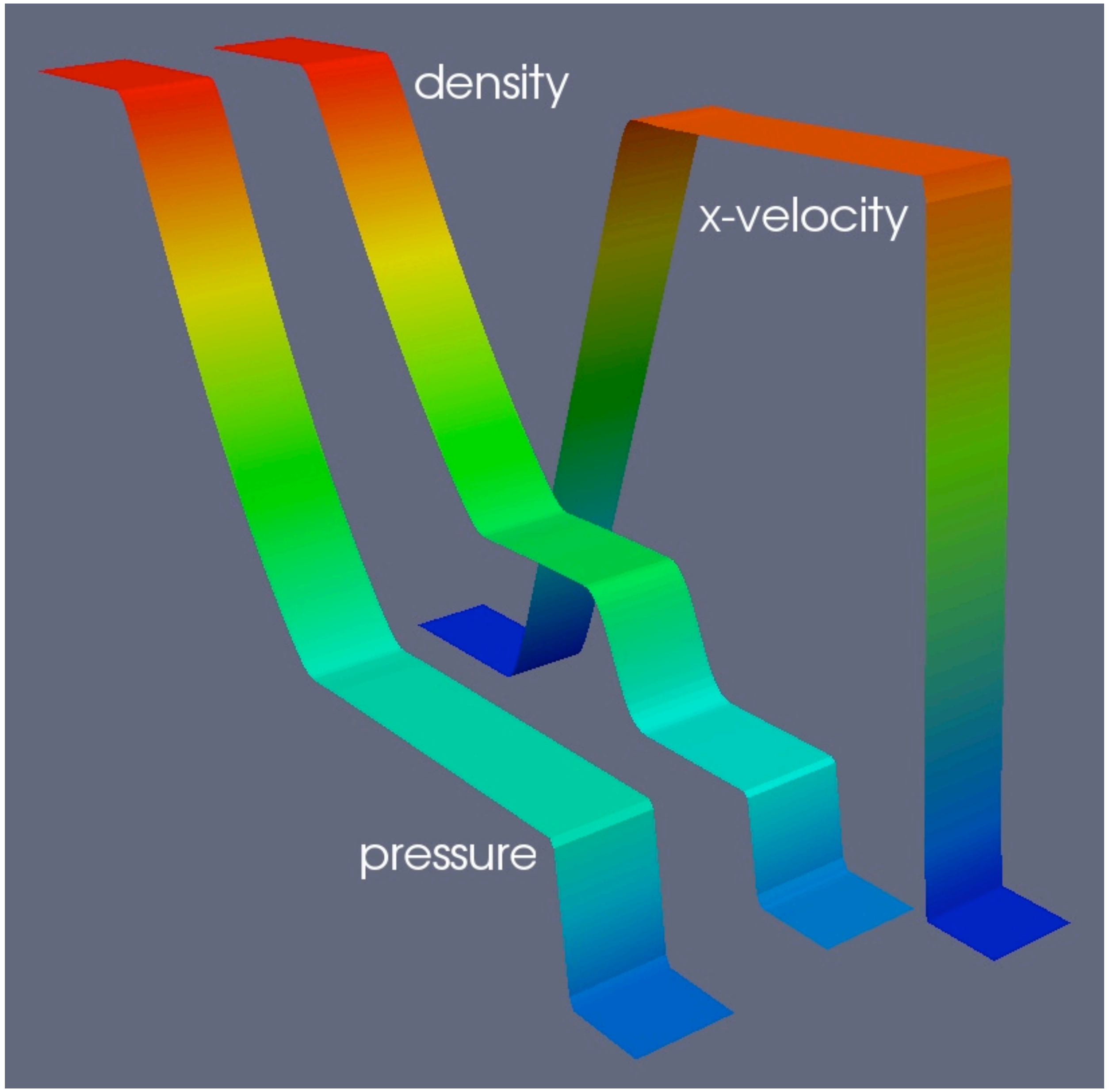
Parallel edge-based assembly



Limiting factors
63 registers and size of shmem per cudathread

Numerical example

- Sod's Shock tube problem in 2D
- Linearized FEM-FCT (density, pressue)
- Artificial dissipation
  - scalar (39 l.o.c.)
  - Roe-type (55 l.o.c.)
- Q1 finite elements
- Regular grid ($\Delta = 8$)
- Greedy coloring ($c = 14$)
- Gcc 4.4.3, CUDA 4.2



Computational efficiency

Computing platforms

- **P1:** Intel Xeon X5680 at 3.33GHz (2x6, no hyperthreading, 2x12MB L3)
  P2: Intel Core i7 at 3.33GHz (1x6, 1x12MB L3) + C2070 (ECC off)
- OpenMP: with 800 edges per „parallel block“
  CUDA: with 64 threads per CUDA block

Comparisons

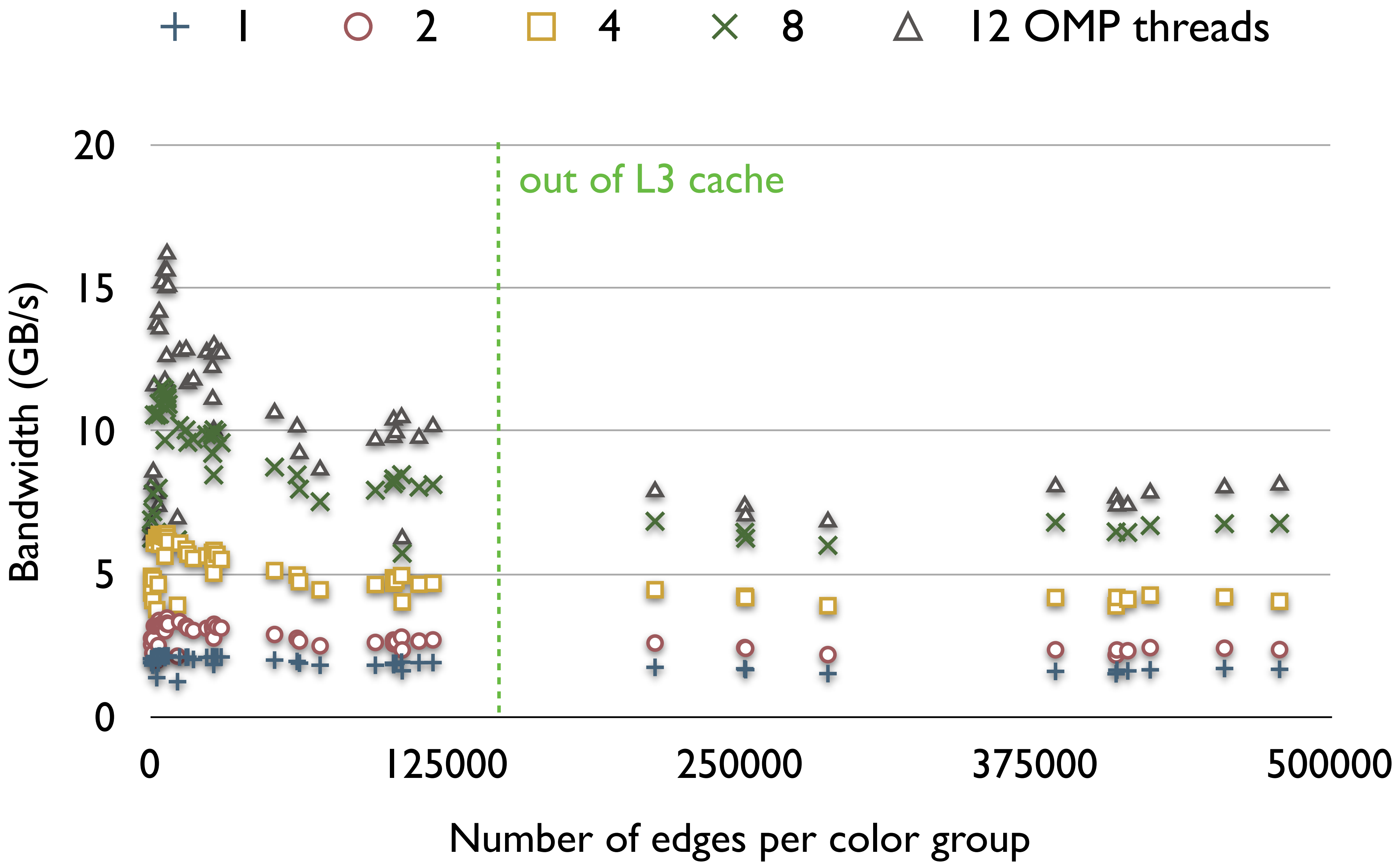
micro benchmark: P1-OpenMP *vs.* P2-CUDA
edge-based vector assembly of a *single* color group

meso benchmark: P2-OpenMP *vs.* P2-CUDA
edge-based vector assembly over all color groups

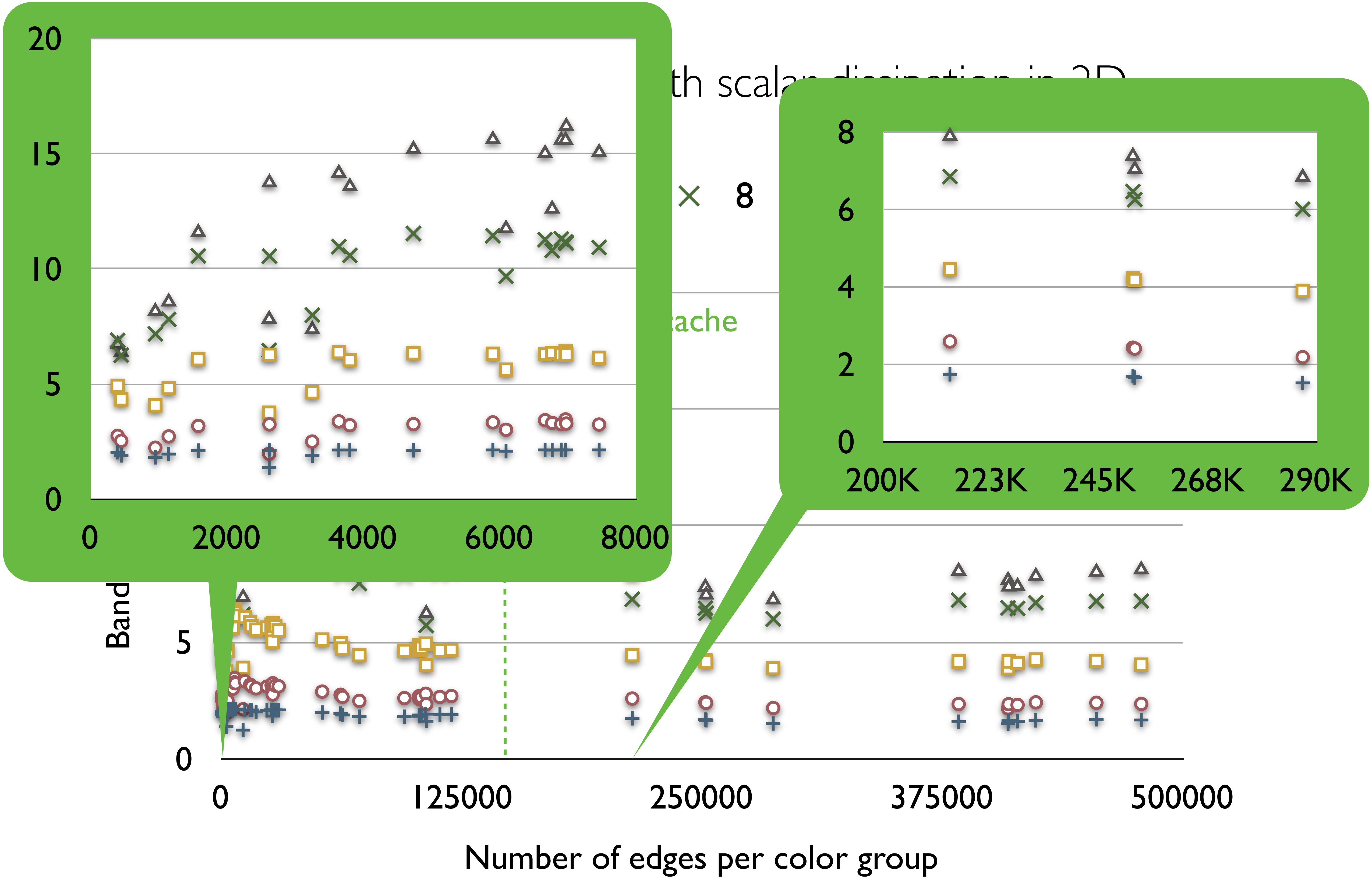
macro benchmark: P2-OpenMP *vs.* P2-CUDA
„full“ simulation (100 time steps) w/o I/O-operations

Bandwidth: CPU implementation (P1)

Kernel: inviscid fluxes with scalar dissipation in 2D

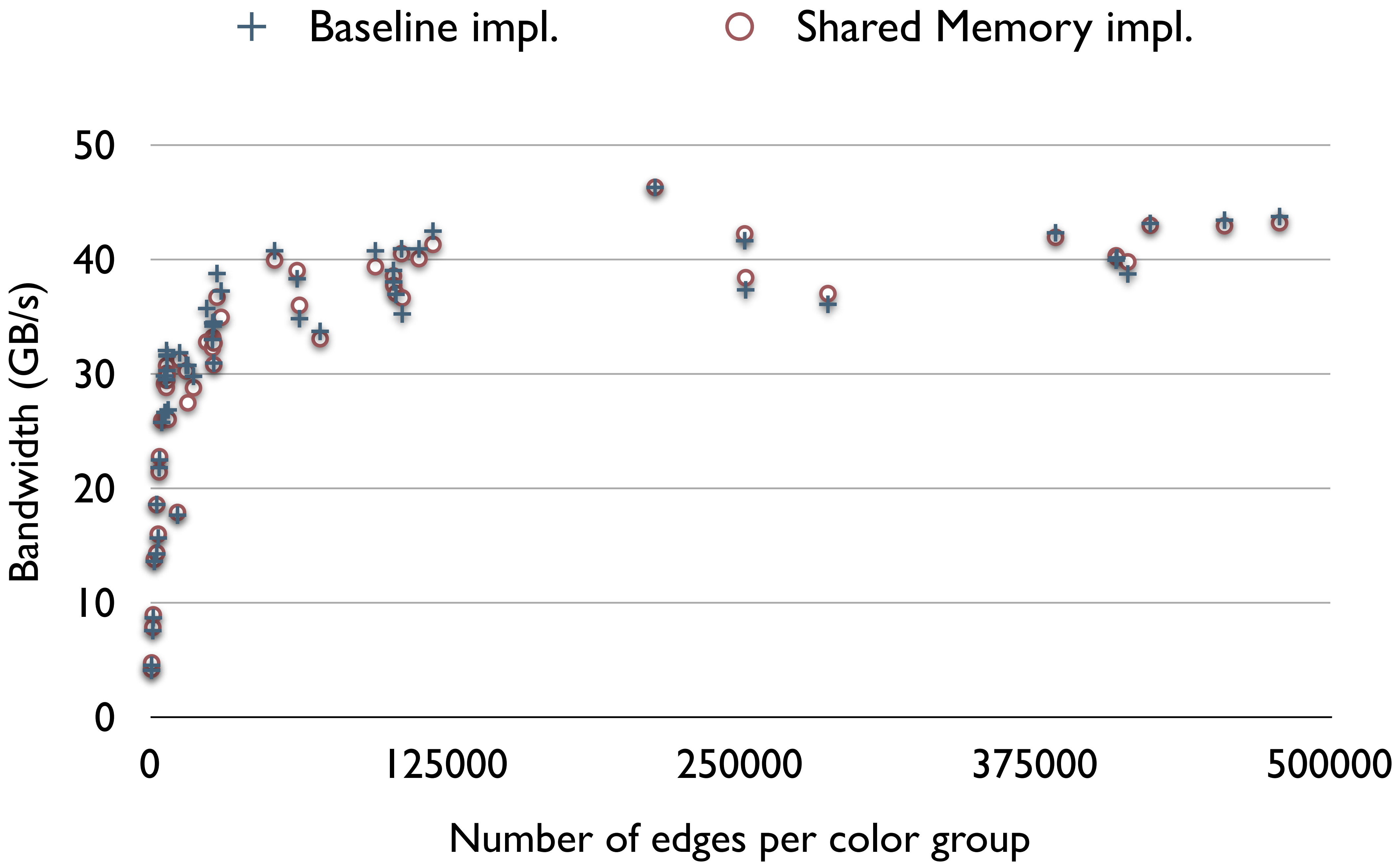


Bandwidth: CPU implementation (P1)



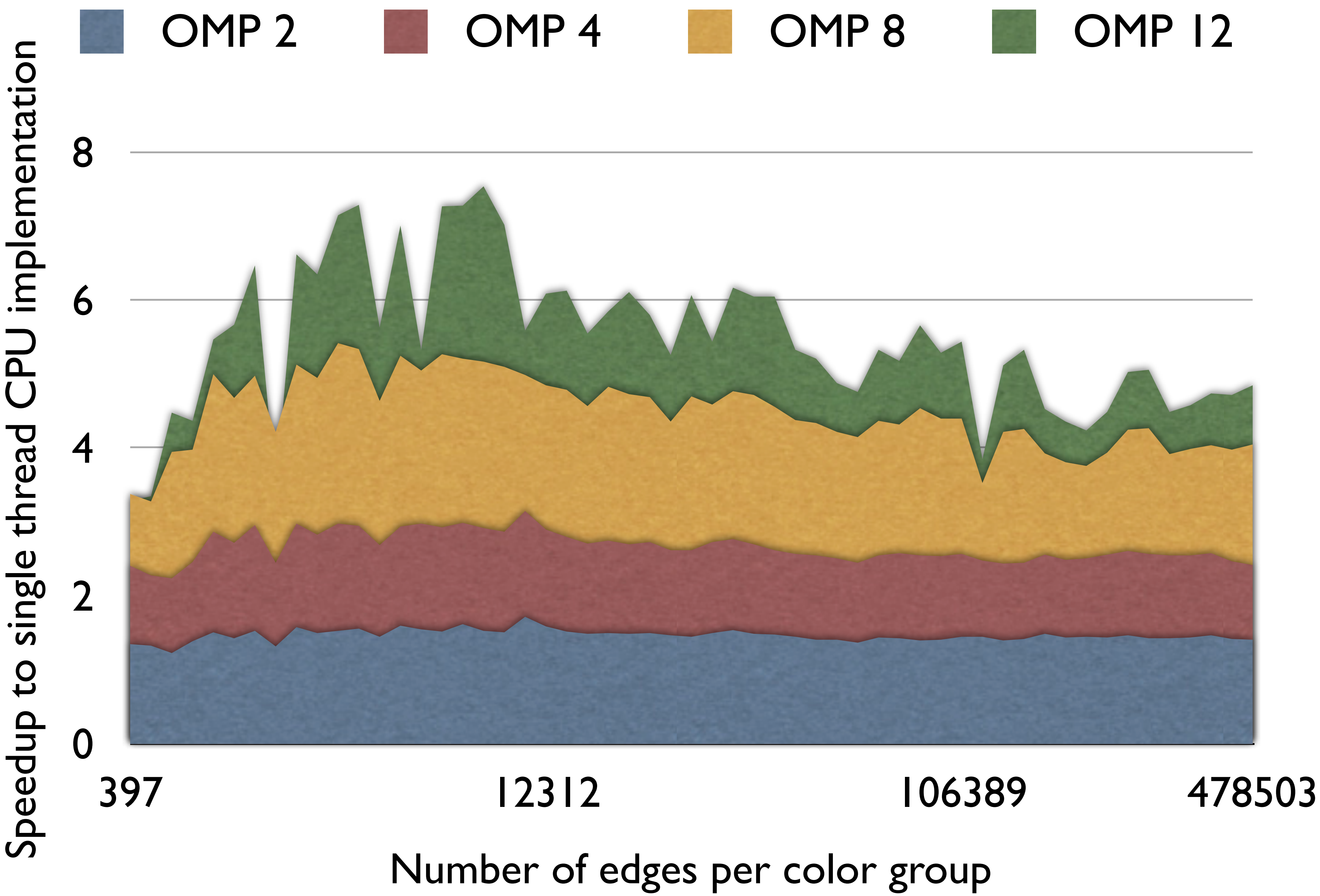
Bandwidth: GPU implementation (P2)

Kernel: inviscid fluxes with scalar dissipation in 2D



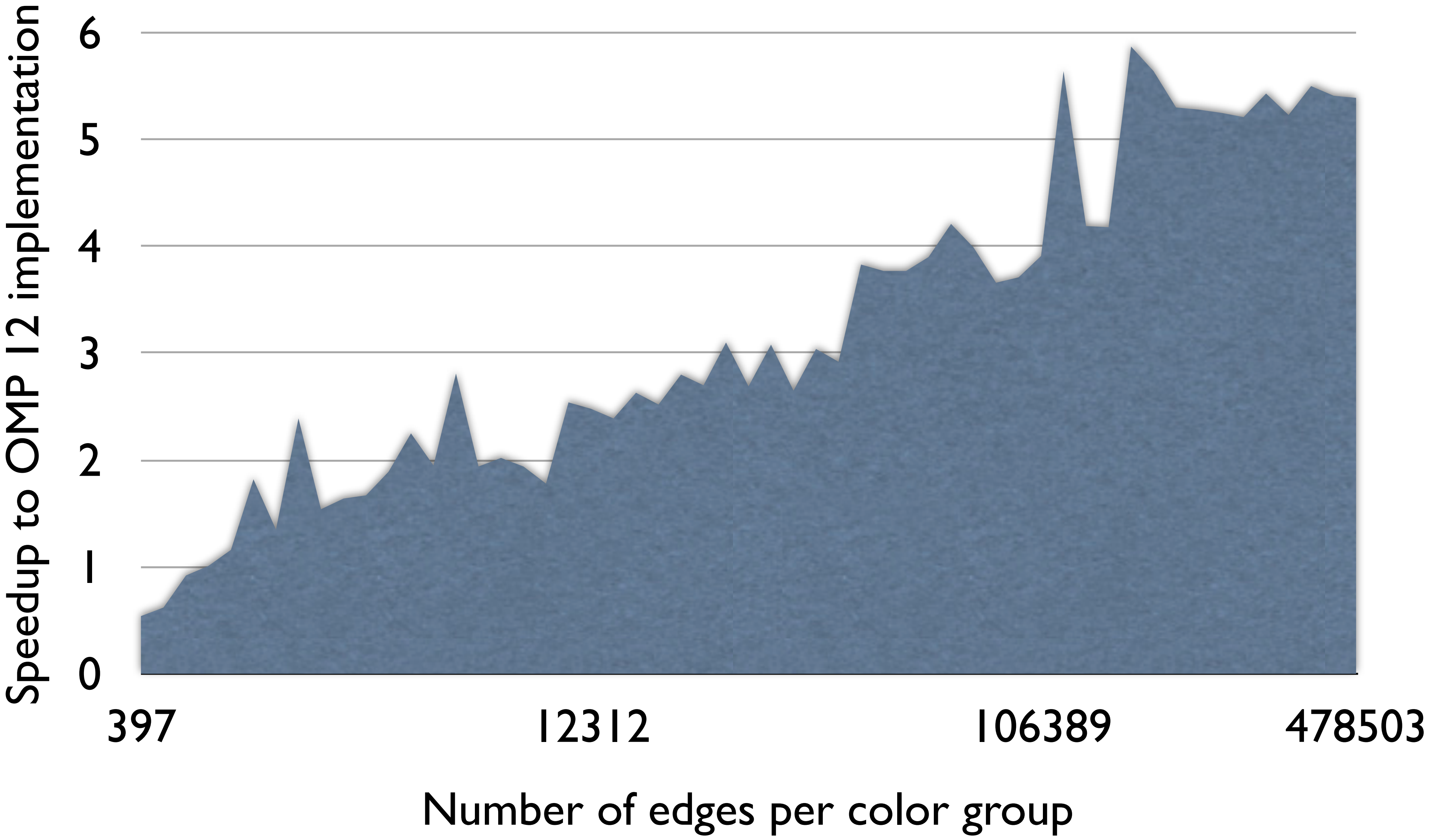
Computing time: CPU implementation (P1)

Kernel: inviscid fluxes with scalar dissipation in 2D



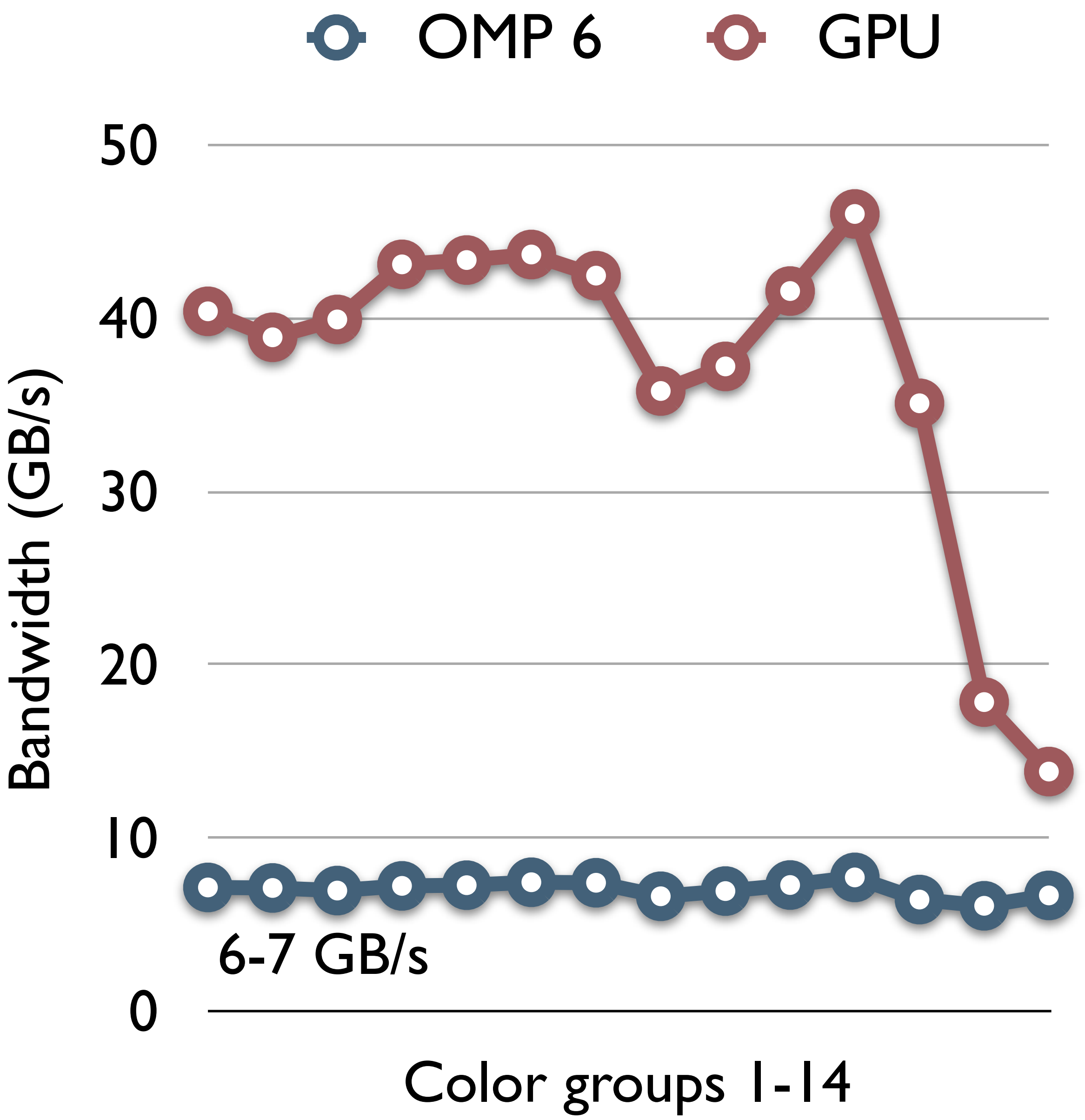
Computing time: GPU implementation (P2)

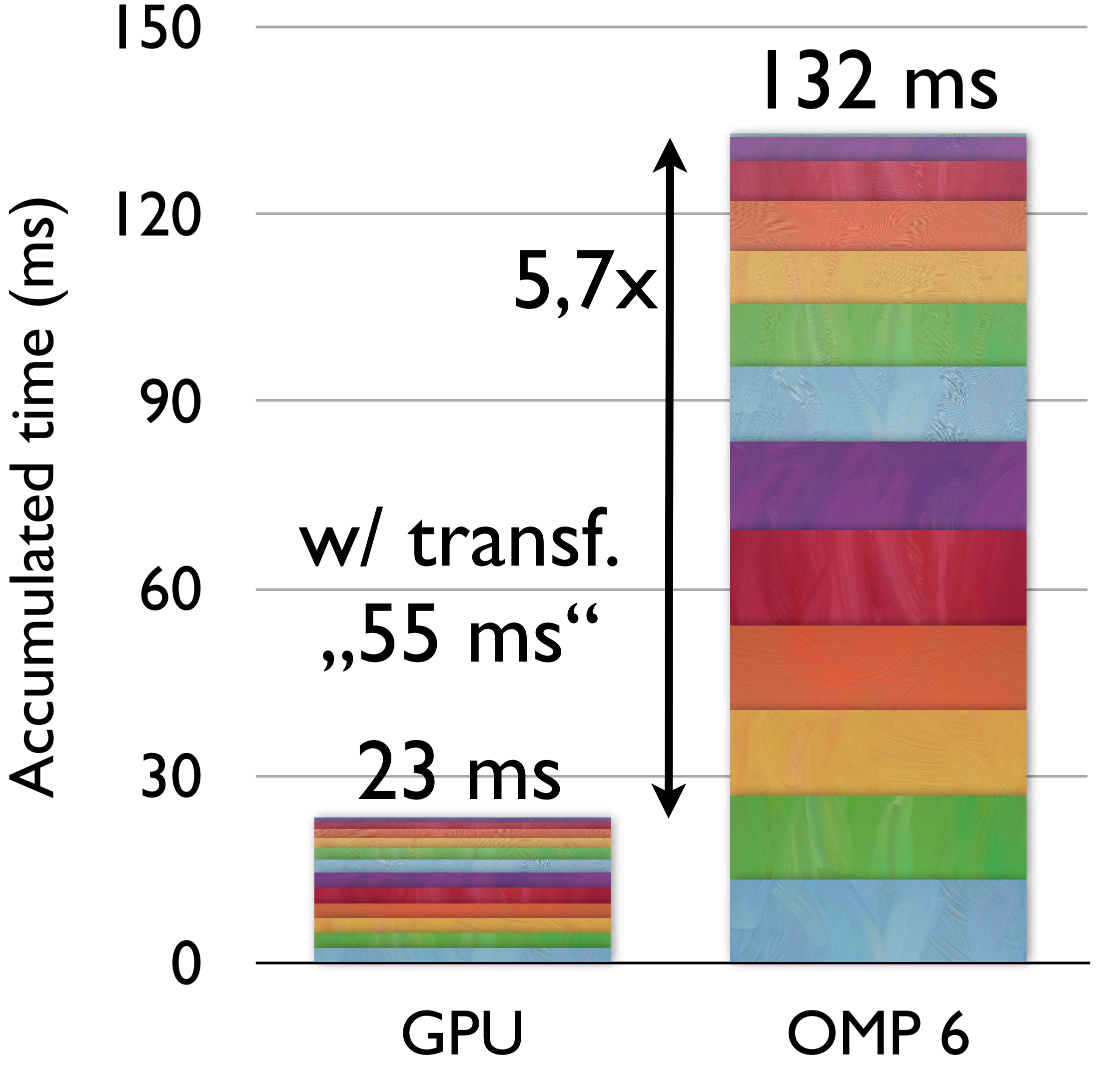
GPU is >5x faster than best CPU implementation



Efficiency of vector assembly (P2)

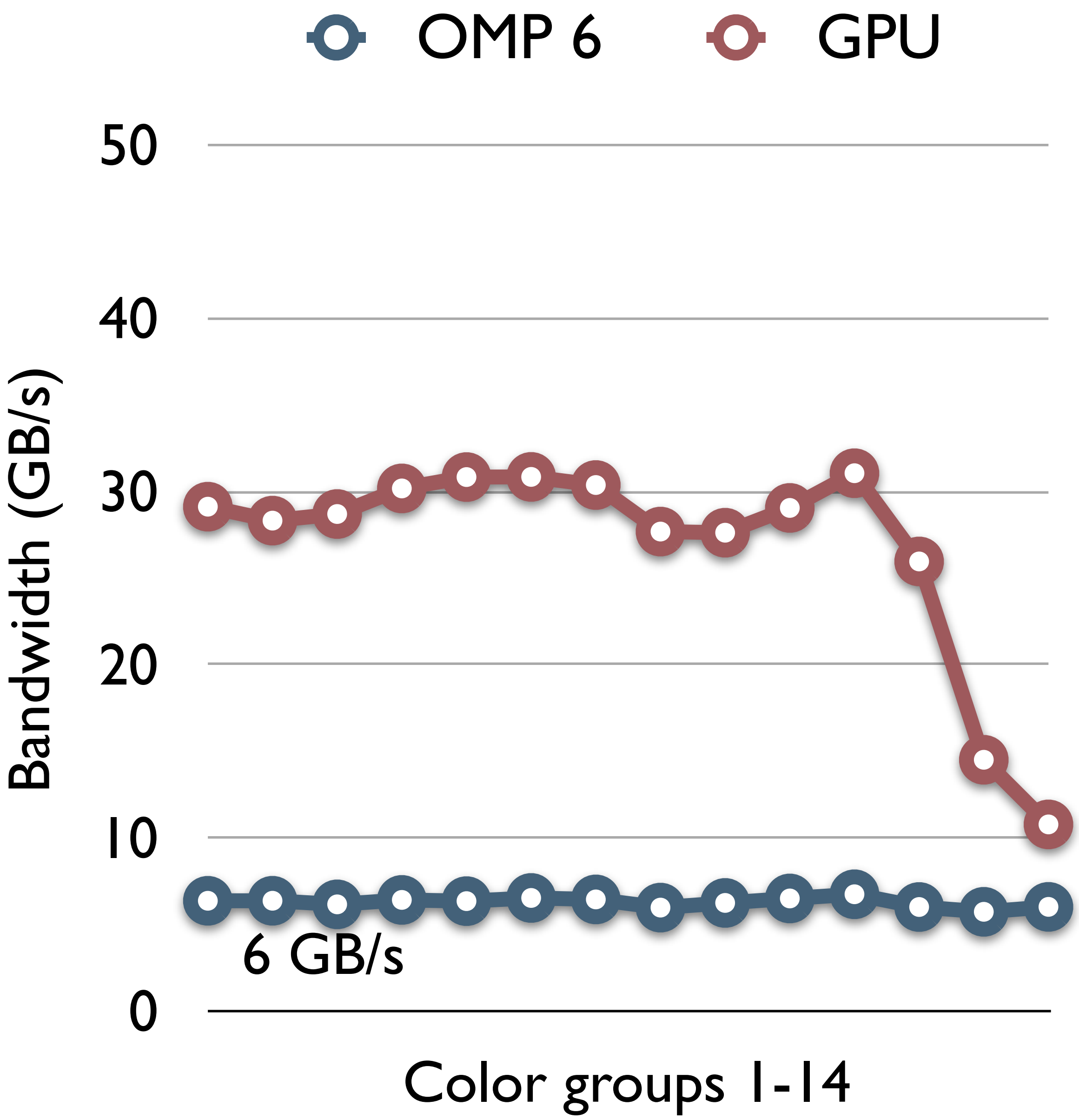
Kernel: inviscid fluxes with scalar dissipation in 2D

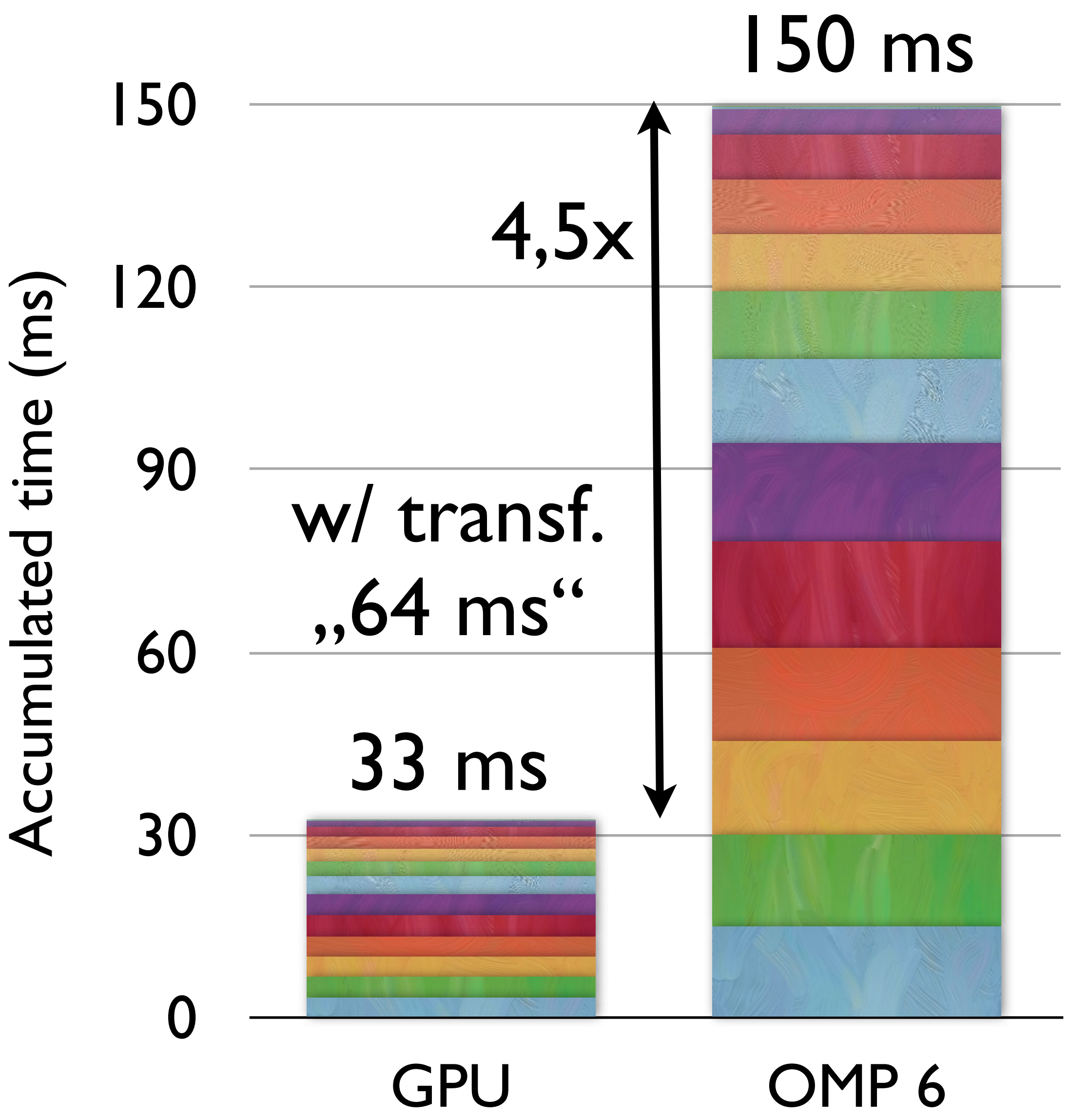




Efficiency of vector assembly (P2)

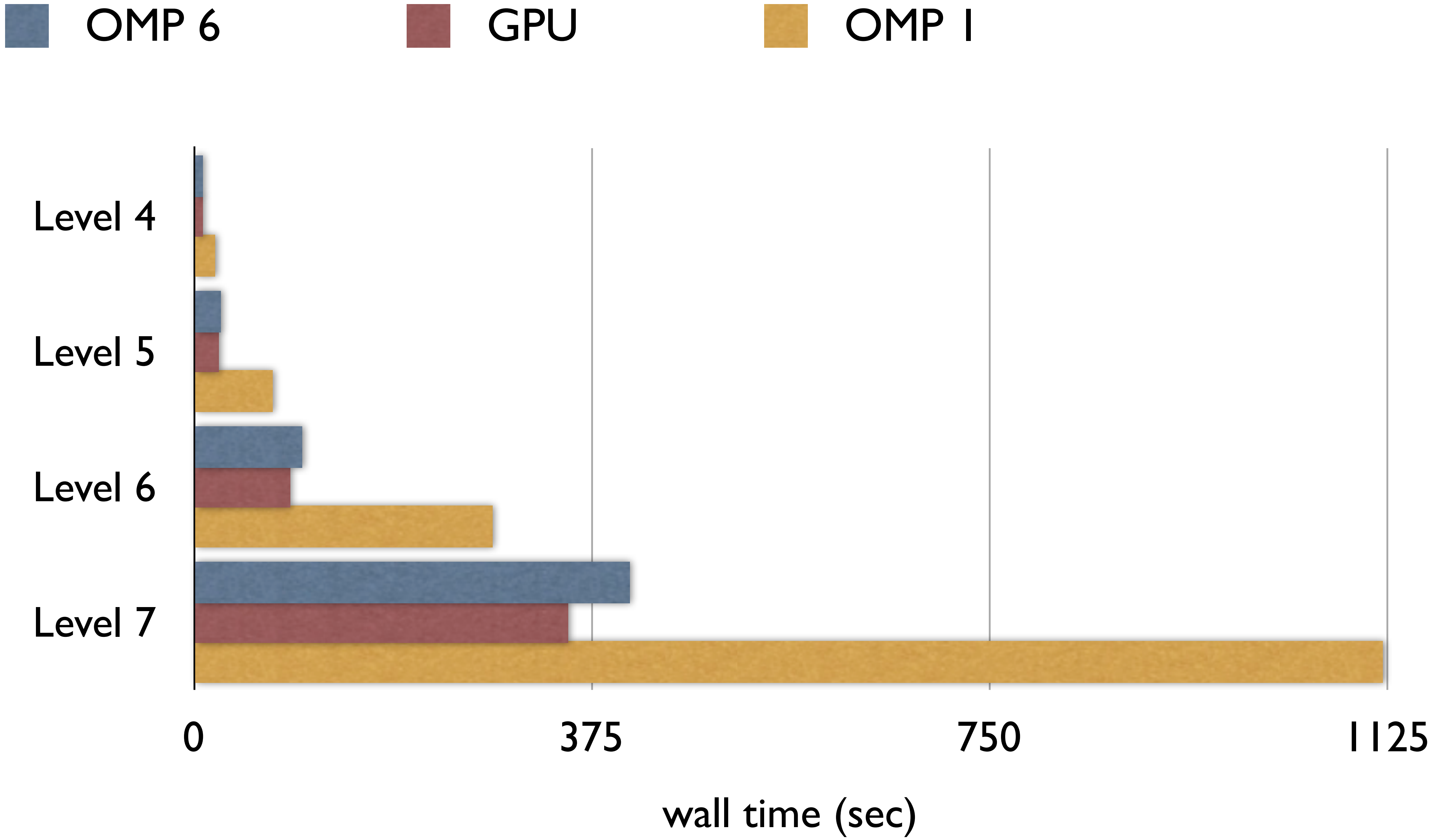
Kernel: inviscid fluxes with register intense Roe-type dissipation in 2D





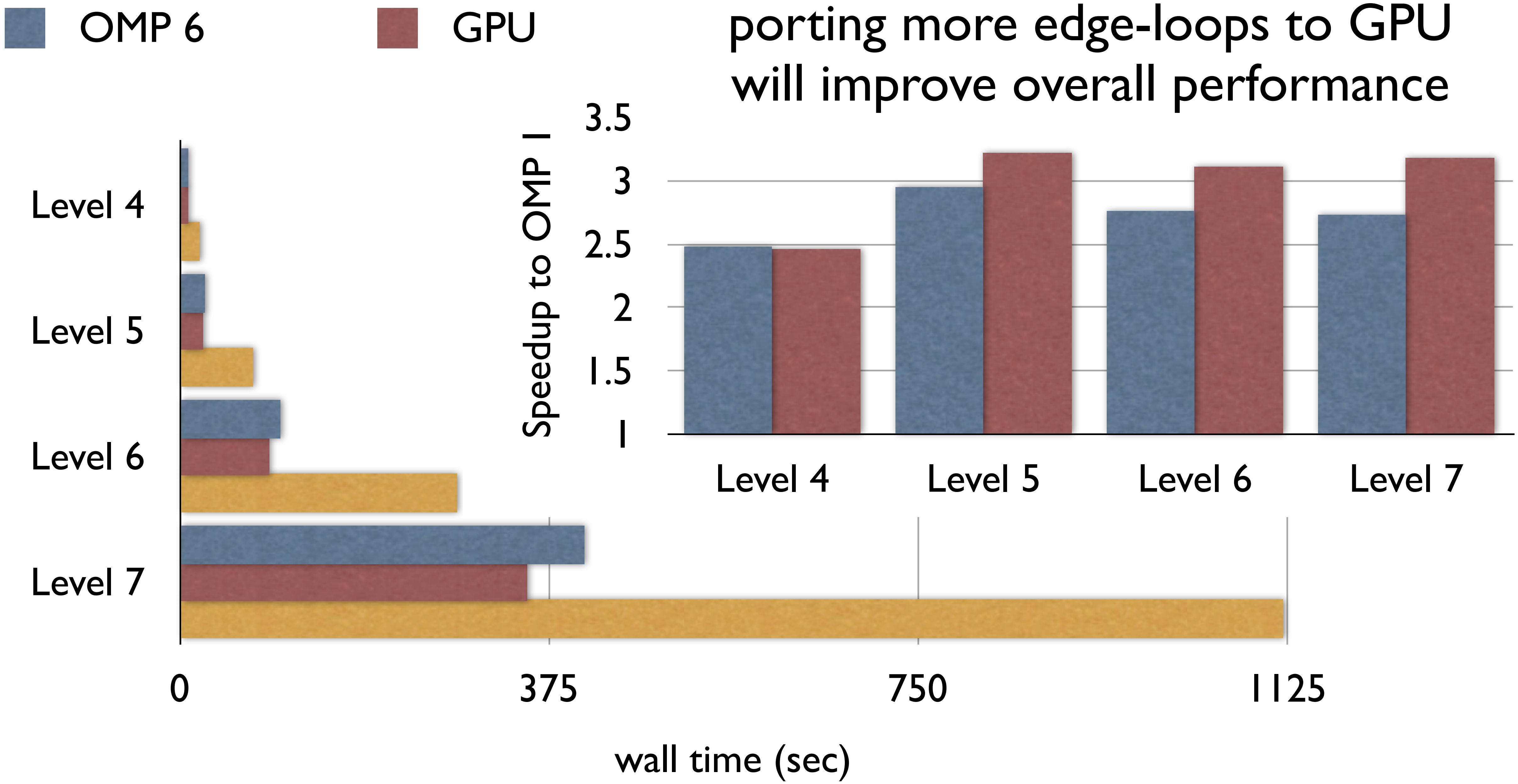
Efficiency of „full“ simulation (P2)

Comparison: full OpenMP *vs.* OpenMP + vector assembly on GPU



Efficiency of „full“ simulation (P2)

Comparison: full OpenMP *vs.* OpenMP + vector assembly on GPU

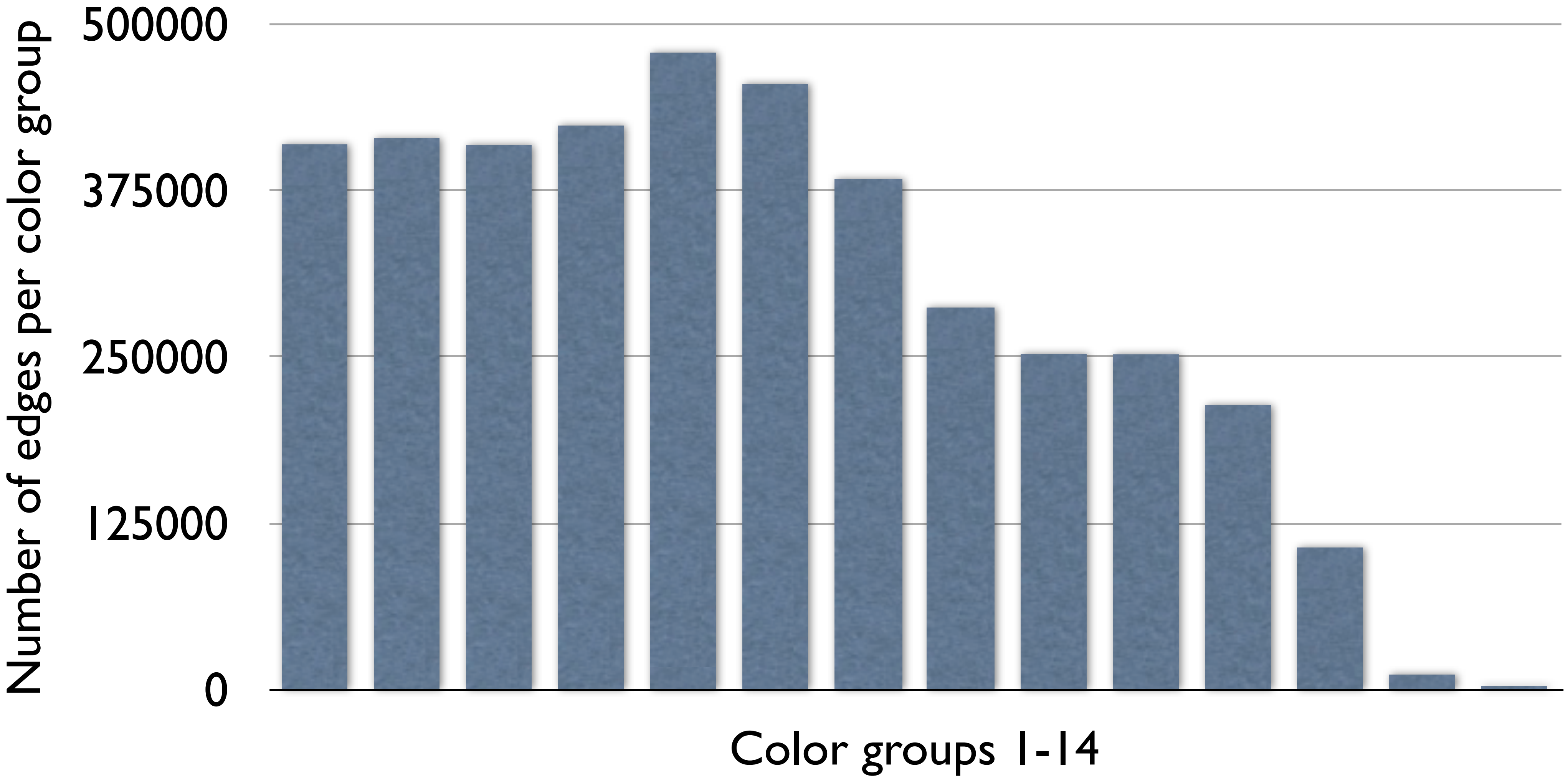


porting more edge-loops to GPU will improve overall performance

A second look on vector assembly

GPU would perform best for only few groups with an equally distributed number of edges per color

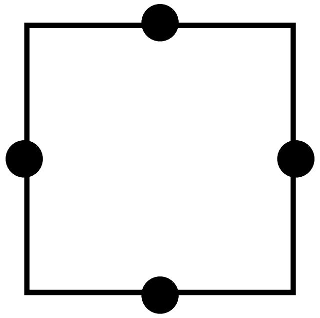
=> use better coloring algorithm and „better“ finite elements (?)

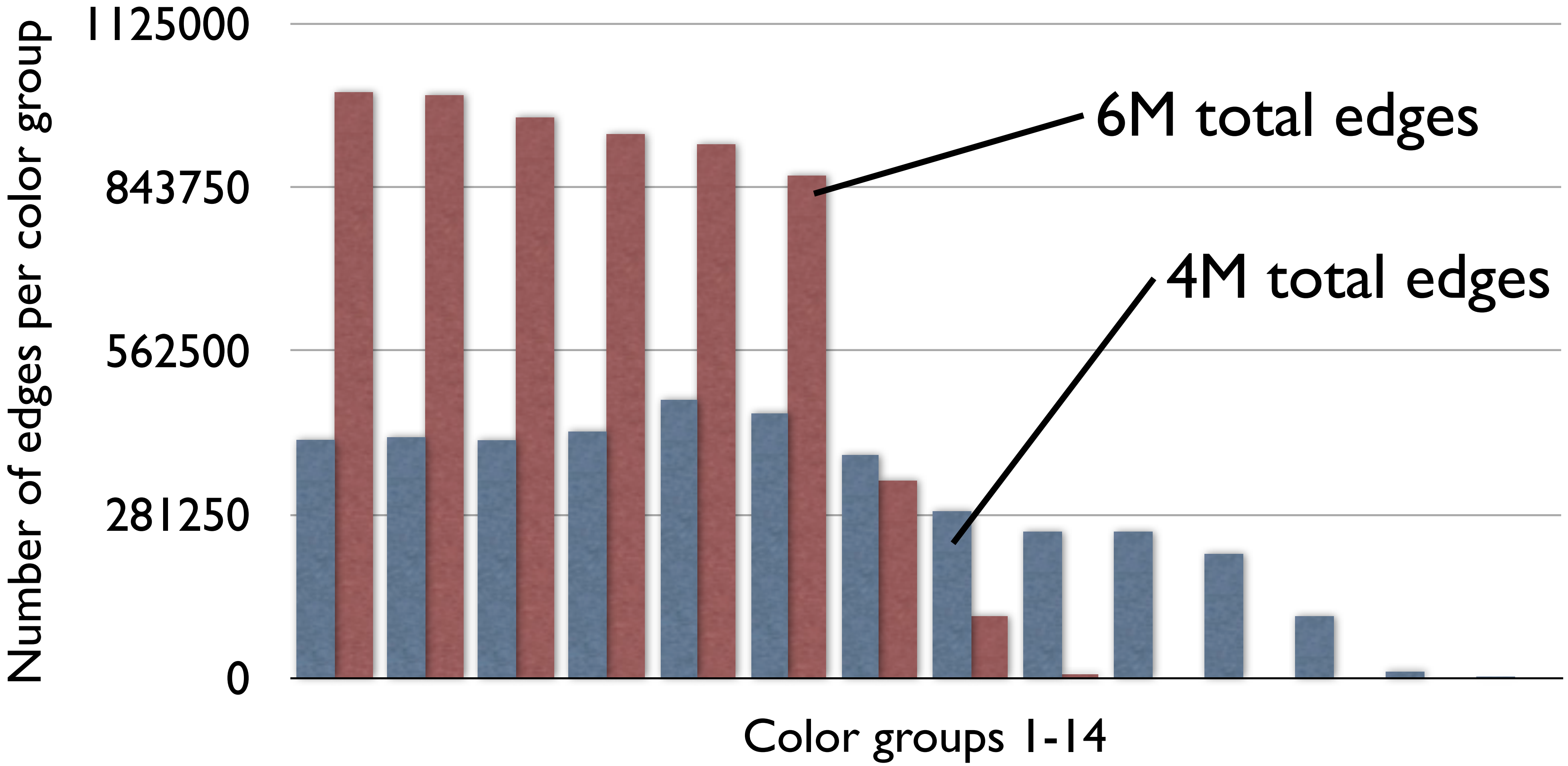


A second look on vector assembly

Non-conforming rotated bilinear Q1~ finite element

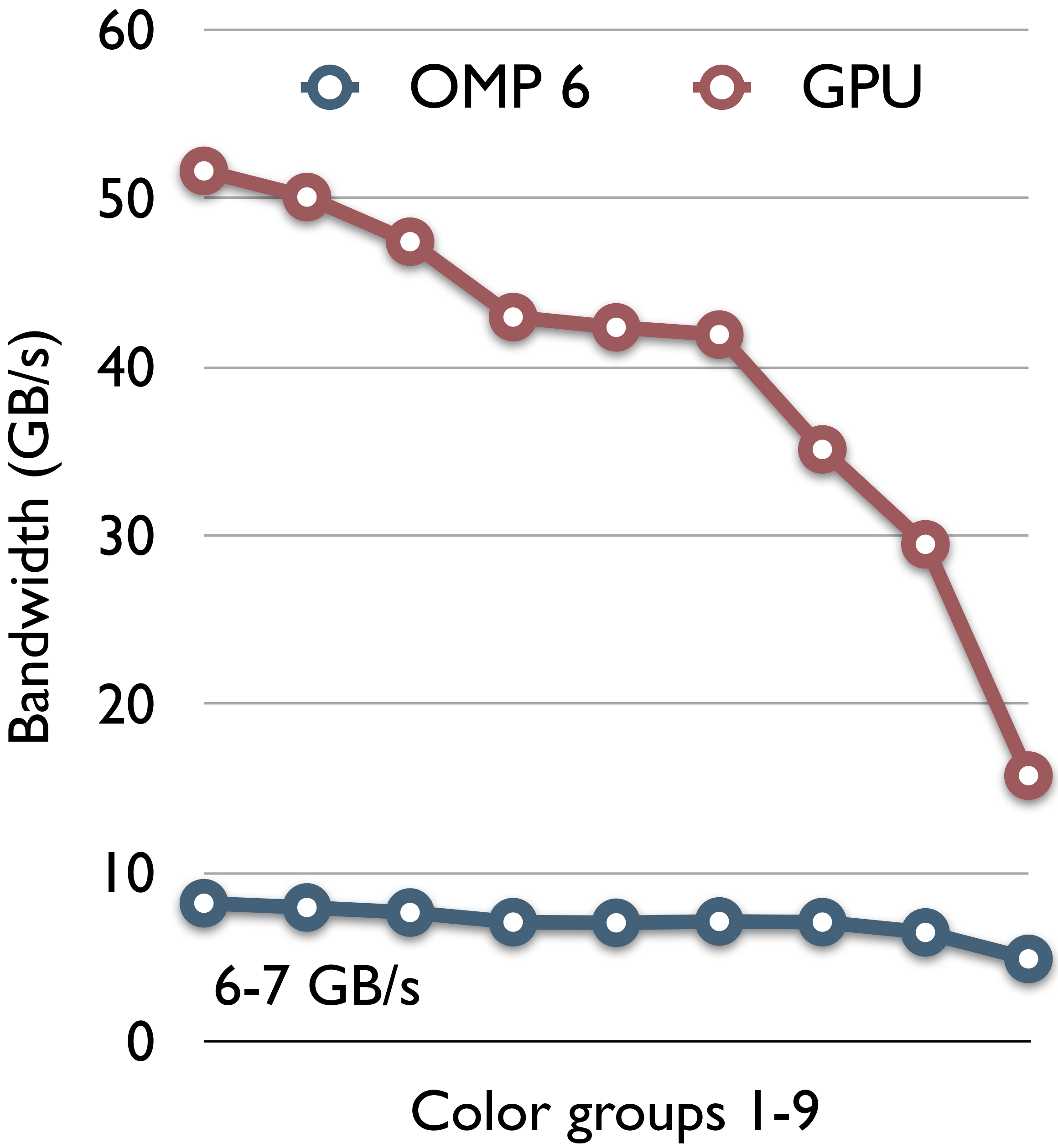
- DOFs are located at the midpoints of edges
- each DOF always couples with 6 neighbors (10 in 3D)

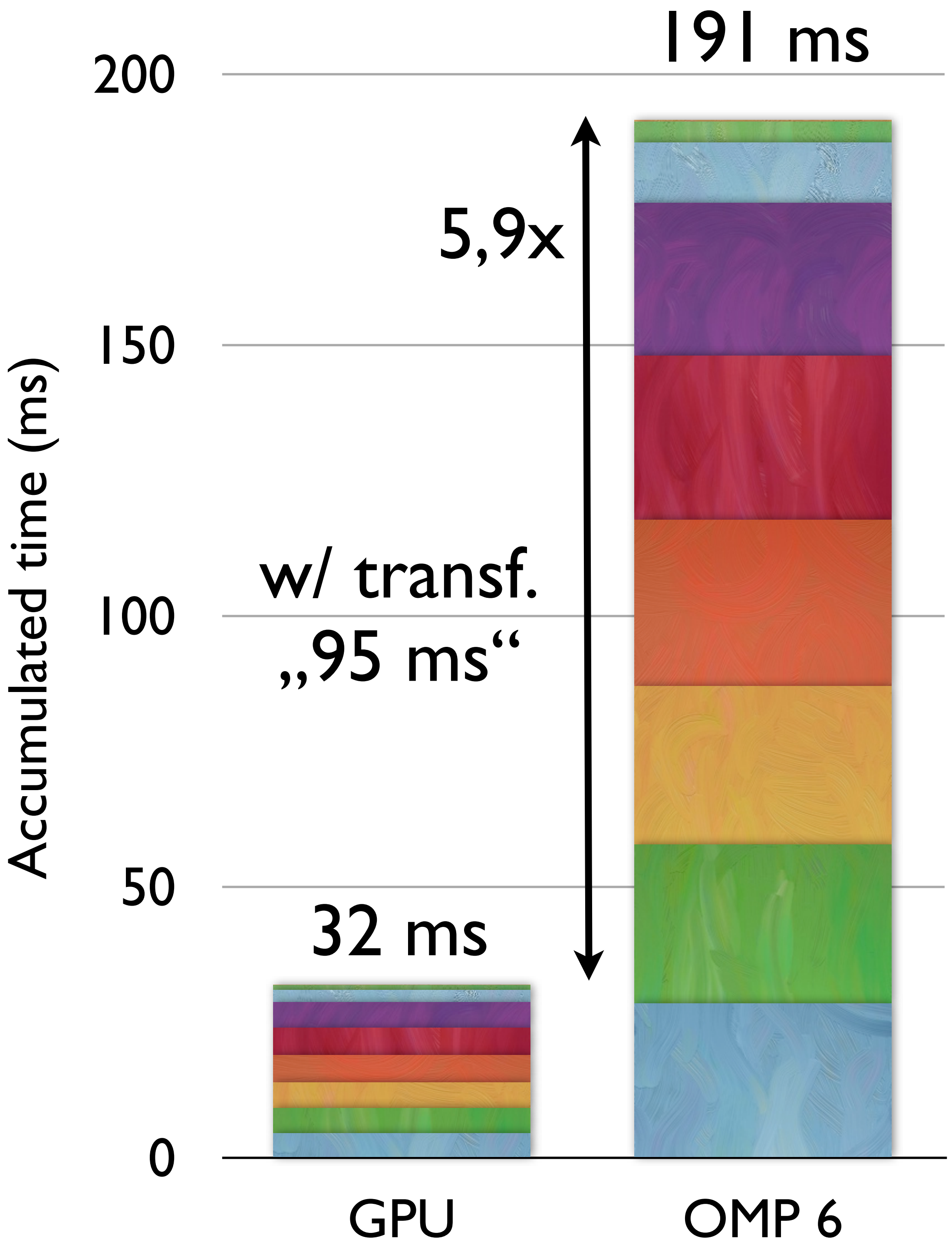




Non-conforming finite elements

Kernel: inviscid fluxes with scalar dissipation in 2D





Technical details

Handle-based storage manager

- (De-)allocate new memory assigned to ihandle (= unique integer)

```
call storage_new( (/a,b,c/), ST_DOUBLE, ihandle, [rheap] )
call storage_free( ihandle, [rheap] )
```

- Associate 3D double pointer to memory at ihandle

```
call storage_getbase( ihandle, p_Darray3D, [rheap] )
```

- Different memory managers can be used ‚under the hood‘

Fortran only `allocate / deallocate`	CUDA `cudaHostAlloc / cudaFreeHost`
🟩 works with all F90 compilers 🟩 no asynchronous transfers	🟩 works with F2003: `iso_c_binding` 🟩 fast and asynchronous transfers

Handle-based storage manager, cont‘d

- Full Fortran 95 functionality: `size`, `shape`, assumed-shape arrays
- Lightwight data structures (= collection of handles)

```
type t_matrix
   integer :: na, neq, ncols
   integer :: h_Da, h_Kcol, h_Kld     <- simple resize in mesh adaptation
```

- Co-Processor support does not break legacy code
- Memory transfers and accessing memory on device

```
storage_syncMem(ihandle, <direction>, async=YES/NO, ...)
storage_getMemPtr(ihandle, p_memptr, ...) -> pass p_memptr to GPU kernel
```

- Support for OpenCL, ... can be integrated easily (!?)

Meta-programming library

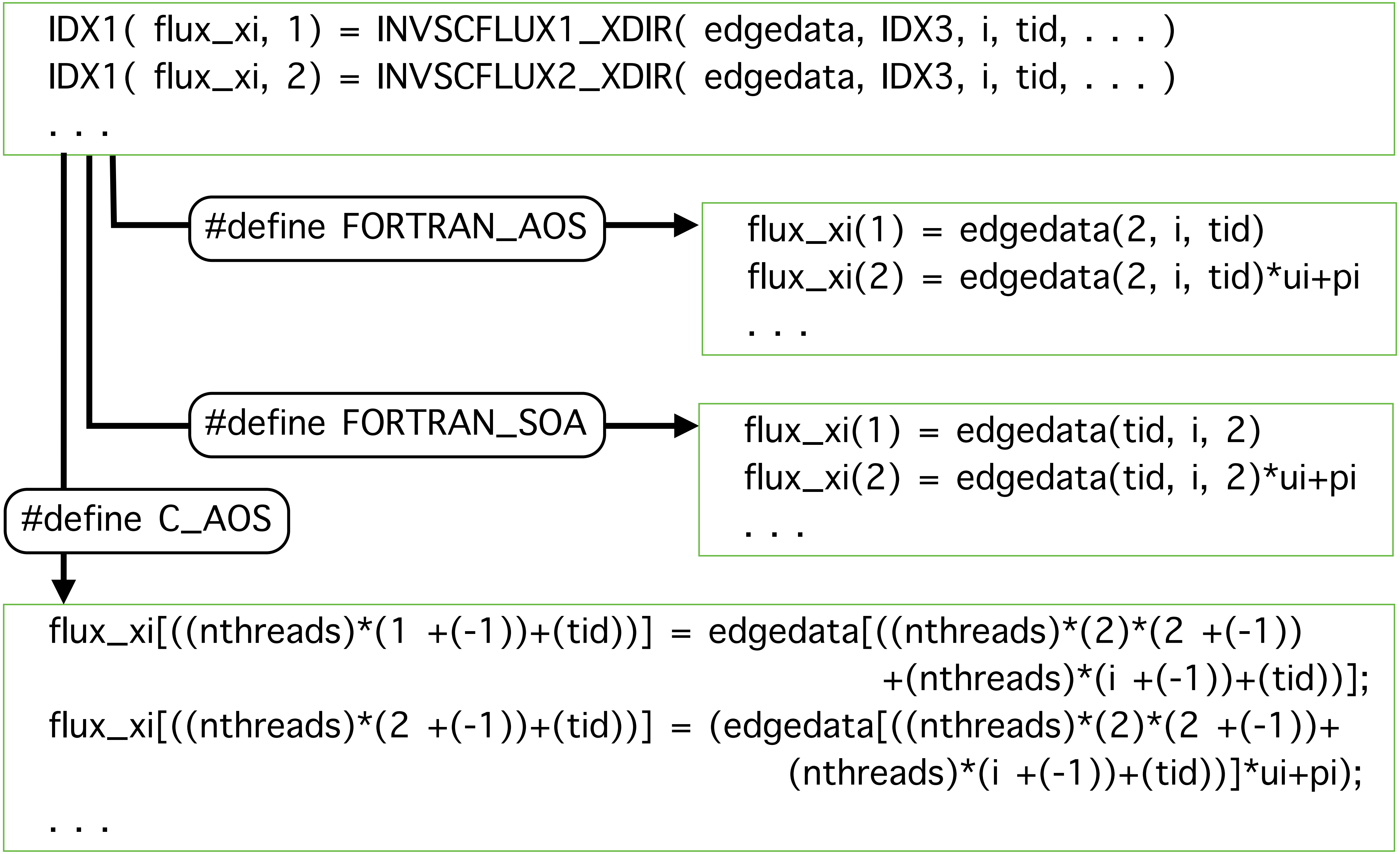
Example: Roe's Riemann solver is the same on CPU/GPU except for

- different programming languages (Fortran/C++)
- different index addressing (0-/1-based)
- different memory layouts (AoS/SoA/mixture)

- Meta-programming of application code via pre-processor macros
  - tedious to find least common subset of built-in pre-processor features supported by all Fortran compilers
  - GNU cpp: F90 → f90 + Fortran compiler

Inspired by presentation by X. Roca at FEF 2011

Meta-programming library



```
IDX1( flux_xi, 1) = INVSCFLUX1_XDIR( edgedata, IDX3, i, tid, . . . )
IDX1( flux_xi, 2) = INVSCFLUX2_XDIR( edgedata, IDX3, i, tid, . . . )
. . .
```

#define FORTRAN_AOS

```
flux_xi(1) = edgedata(2, i, tid)
flux_xi(2) = edgedata(2, i, tid)*ui+pi
. . .
```

#define FORTRAN_SOA

```
flux_xi(1) = edgedata(tid, i, 2)
flux_xi(2) = edgedata(tid, i, 2)*ui+pi
. . .
```

#define C_AOS

```
flux_xi[((nthreads)*(1 +(-1))+(tid))] = edgedata[((nthreads)*(2)*(2 +(-1))
                                        +(nthreads)*(i +(-1))+(tid))];
flux_xi[((nthreads)*(2 +(-1))+(tid))] = (edgedata[((nthreads)*(2)*(2 +(-1))+
                                    (nthreads)*(i +(-1))+(tid))]*ui+pi);
. . .
```

Summary

Parallelization of edge-based CFD-solver Featflow2

- Group-FEM formulation leads to memory and time efficient edge-based assembly on Many-Core architectures
- Minimally invasive integration of GPU acceleration in legacy code
- Meta-programming library simplifies mixing of programming languages and enables reuse of application code

Future plans

- Explore benefits of non-conforming FEs for edge-parallelization
- Port more edge-loops to CUDA and add multi-GPU support
- Combine assembly on CPU and GPU adaptively

References

- C.A.J. Fletcher, *The group finite element formulation.* CMAME 1983, 37(2), pp. 225–244.
- D. Kuzmin, *Explicit and implicit FEM-FCT algorithms with flux linearization*. JCP 2009, 228(7), pp. 2517–2534.
- D. Kuzmin, M. M, S. Turek, *Multidimensional FEM-FCT schemes for arbitrary time-stepping*. IJNMF 2003, 42(3), pp. 265-295.
- D. Kuzmin, M. M, J.N. Shadid, M. Shashkov, *Failsafe flux limiting and constrained data projections for equations of gas dynamics*. JCP 2010, 229(23), pp. 8766–8779.
- X. Roca, private communication (FEF 2011, Munich)