Science and Technology Facilities Council

epcc | archer2

Cross-platform programming model for GPU implementation of OpenFOAM using only ISO C++

Science and Technology Facilities Council
Hartree Centre

Jony Castagna
Raynold Tan

Science and Technology Facilities Council
Scientific Computing

Mayank Kumar
Wendi Liu

University of Exeter

Gavin Tabor

OpenFOAM®
KEYSIGHT

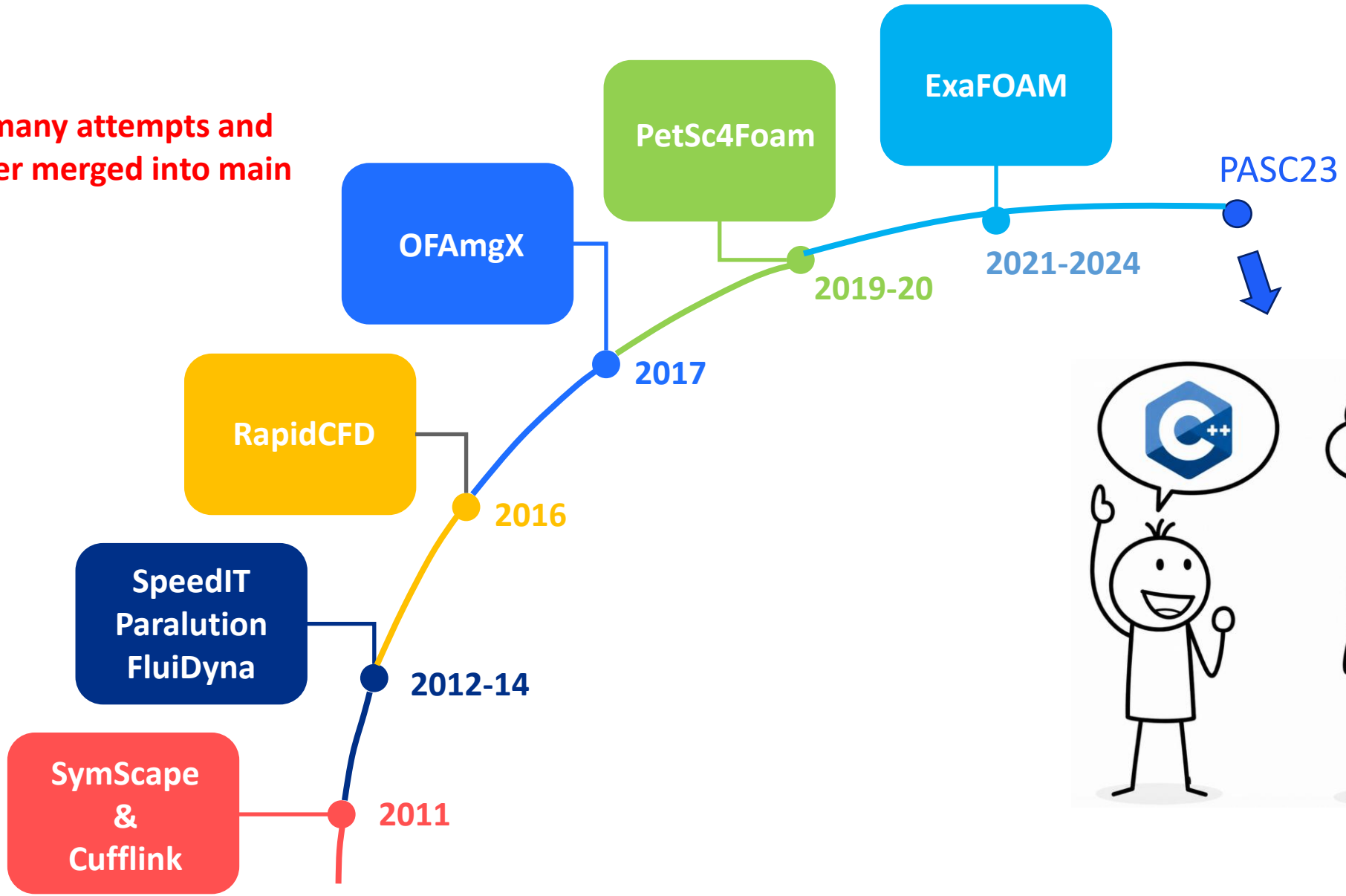
Mattijs Janssens
Andrew Heather

Outline

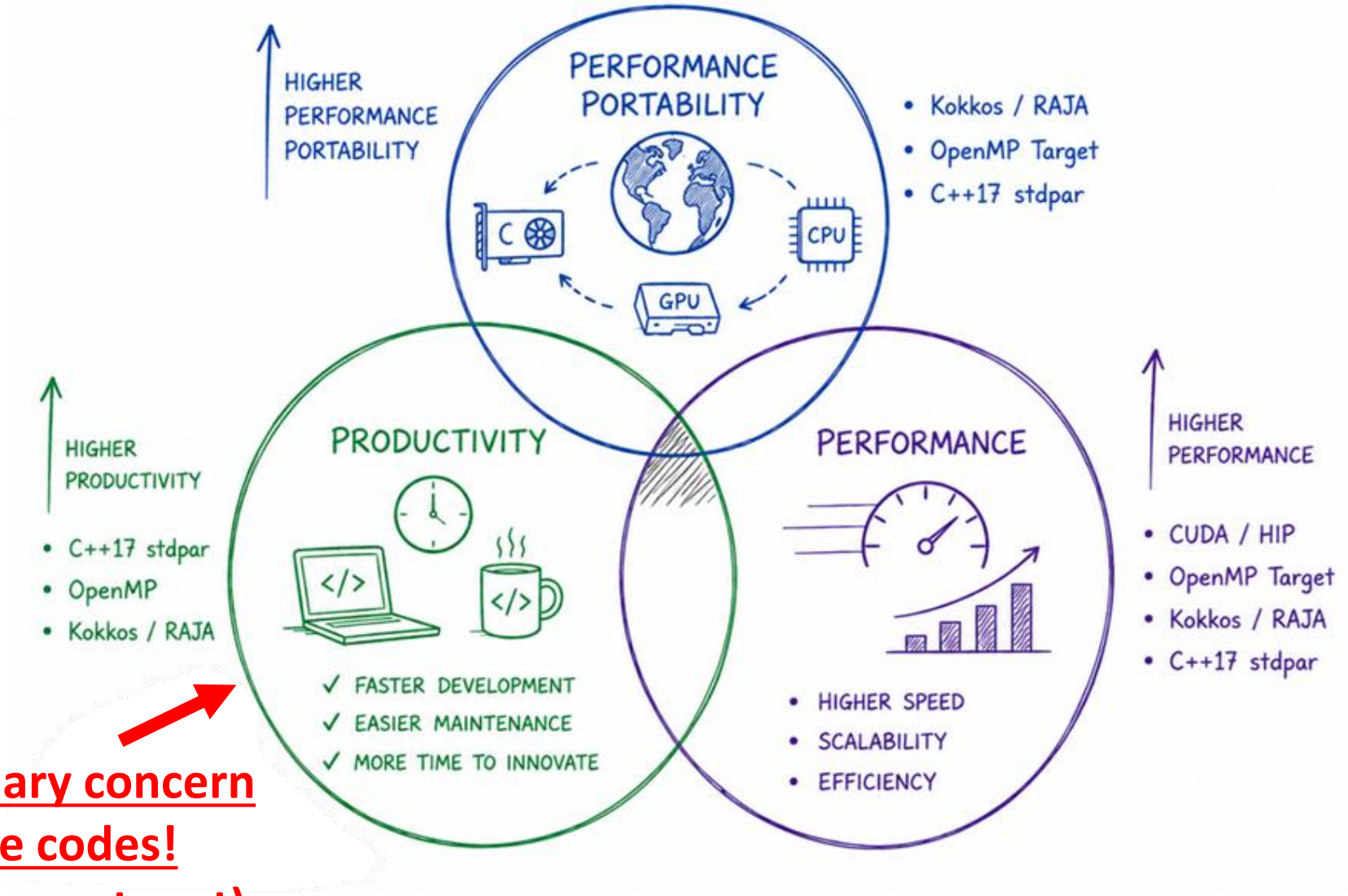
- A once in a life opportunity!
- How we did it?
- Current performance
- The magic of UMPIRE!
- How to use it?
- Still to do… (a lot)!

Timeline GPU implementation for OpenFOAM

Why so many attempts and why never merged into main release?



Performance, Portability, Productivity



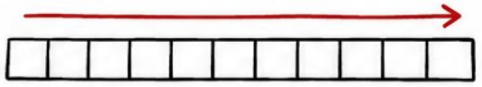
Often the primary concern in very large codes! (lower development cost)

C++17 offloading

1 Classical C++ Loop

```
void saxpy(std::vector<double>& y,
           const std::vector<double>& x,
           double a)
{
    size_t n = x.size();

    for (size_t i = 0; i < n; ++i) {
        y[i] += a * x[i];
    }
}
```

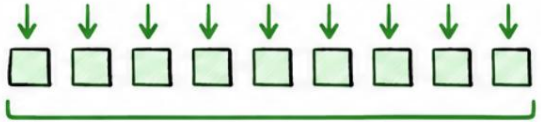
Sequential execution
One element at a time

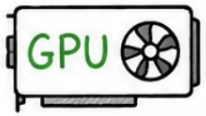


2 C++17 Loop Offloaded on GPU via std::execution

```
#include <execution>
#include <vector>

void saxpy(std::vector<double>& y,
           const std::vector<double>& x,
           double a)
{
    size_t n = x.size();

    std::transform(std::execution::par_unseq,
                   x.begin(), x.end(),
                   y.begin(), y.begin(),
                   [a](double xi, double yi) {
                       return yi + a * xi;
                   });
}
```

Parallel execution on GPU
Many elements at the same time





NVIDIA: https://developer.nvidia.com/blog/developing-accelerated-code-with-standard-language-parallelism/
AMD: https://gpuopen.com/learn/amd-lab-notes/amd-lab-notes-hipstdpar-readme/

How we did it?

We started from:

icoFoam solver

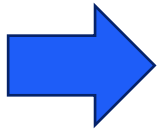
cavity3D

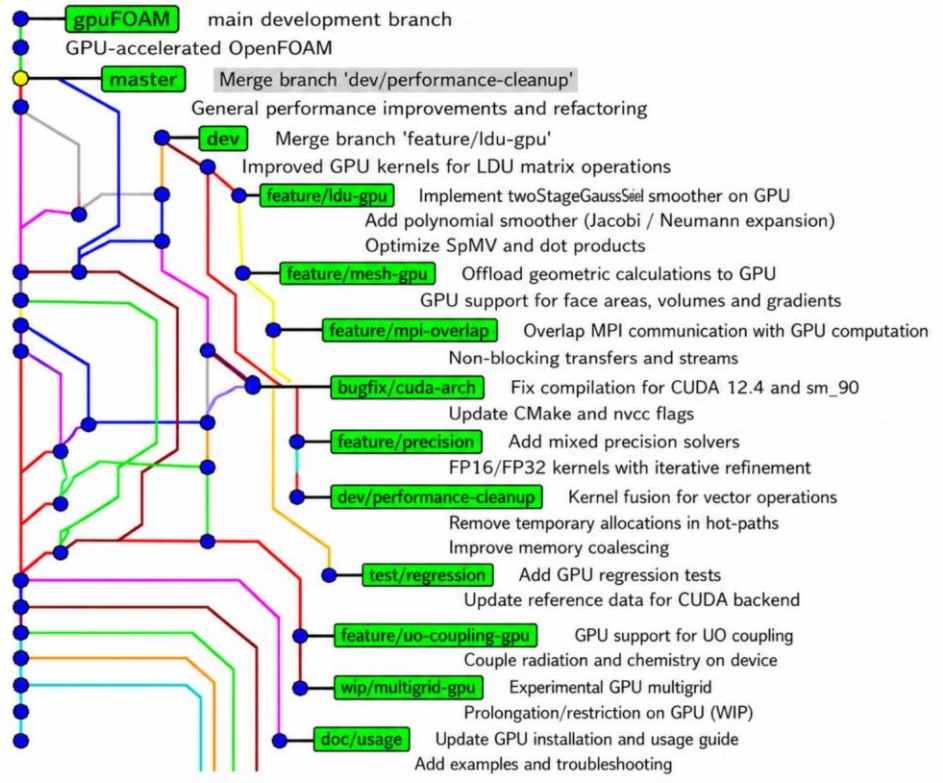
PCG

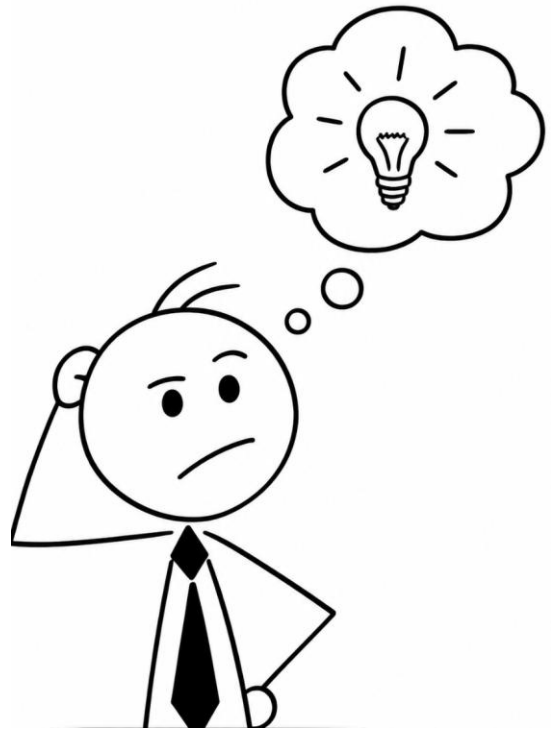
Atmul

Interpolation

Finite Volume operations







A spaghetti git repo…

We need to rewrite it all!

1. Primitives
   - 1.a scalar
   - 1.b tensor
2. Containers
   - 2.a Lists
   - 2.b dimensionSet
3. Fields
   - 3.a scalarField
   - 3.b tensorField
   - 3.c geometricField
   - 3.d DynamicField
   - 3.e TFOR_ALL
4. Matrix
   - 6.a lduMatrix
   - 6.b lduMesh
   - 6.c scalarMatrix
5. Solvers
   - 7.a PCG
   - 7.b PBiCG
   - 7.c PBiCGStab
   - 7.c Smoother
   - 7.c GAMGv
6. FiniteAera
7. FiniteVolume
   - 5.a fvMatrix
   - 5.b SurfaceIntegrate
   - 5.c Interpolation
   - 5.d Gradient
   - 5.e fvPatchFields
   - 5.f fvMesh
8. Turbulence
   - 8.a Base
   - 8.b kOmegaSST
   - 8.c omegaWallFunction
9. Utilities
   - 9.a MeshGeneration
   - 9.b thermoTools

Principle: monotonically increase GPU offloading capabilities!

Easier to debug, expand, compare

Smooth integration into CPU version

We count ~200 **par_unseq (via apply_policy)** over the full OpenFOAM code so far…

However: you should build something **3 times** to get it right! (Uncle Walter)

Different levels of integration

Level A: same code section for CPU and GPU, only parallel execution directive differs

OpenFOAM Loop

```
#define TFOR_ALL_F_OP_FUNC_F_F
(typeF1, f1, OP, FUNC, typeF2, f2, typeF3, f3)
{
    /* Check fields have same size */
    checkFields(f1,f2,f3,"f1 "#OP" "#FUNC"(f2,f3)");

    /* Field access */
    List_ACCESS(typeF1, f1, f1P);
    List_CONST_ACCESS(typeF2, f2, f2P);
    List_CONST_ACCESS(typeF3, f3, f3P);

    /* Loop: f1 OP FUNC(f2, f3) */
    const label loop_len = (f1).size();

    for (label i = 0; i < loop_len; ++i)
    {
        (f1P[i]) OP FUNC((f2P[i]), (f3P[i]));
    }
}
```

ISO C++ Parallel STL Loop

```
#include <algorithm>
#include <execution>

#define TFOR_ALL_F_OP_FUNC_F_F
(typeF1, f1, OP, FUNC, typeF2, f2, typeF3, f3)
{
    /* Check fields have same size */
    checkFields(f1,f2,f3,"f1 "#OP" "#FUNC"(f2,f3)");

    /* Field access */
    List_ACCESS(typeF1,  f1, f1P);
    List_CONST_ACCESS(typeF2, f2, f2P);
    List_CONST_ACCESS(typeF3, f3, f3P);

    const std::size_t n = (f1).size();

    /* Parallel index range [0, n) */
    auto indices = std::views::iota(std::size_t(0), n);

    std::for_each(
        std::execution::par_unseq,
        indices.begin(),
        indices.end(),
        [&](std::size_t i)
        {
            (f1P[i]) OP FUNC((f2P[i], f3P[i]));
        }
    );
}
```

The body of the loop stays the same!!

Different levels of integration

Level B: switch between loop over faces into loop over cells to avoid race conditions

icoFoam

PCG

$$r_0 = b - Ax_0$$

$$A^T \cdot \psi = D \cdot \psi + L^T \cdot \psi + U^T \cdot \psi$$

```
for (label face=0; face<nFaces; face++)
{
    ApsiPtr[uPtr[face]] += lowerPtr[face]*psiPtr[lPtr[face]];
    ApsiPtr[lPtr[face]] += upperPtr[face]*psiPtr[uPtr[face]];
}
```

Different levels of integration

Level B: switch between loop over faces into loop over cells to avoid race conditions

```
for (label face=0; face<nFaces; face++)
{
    ApsiPtr[uPtr[face]] += lowerPtr[face]*psiPtr[lPtr[face]];
    ApsiPtr[lPtr[face]] += upperPtr[face]*psiPtr[uPtr[face]];
}
```

stdpar
+
atomic
(double precision)

supported on latest GPUs, but requires atomic primitives objects

rewrite as loops without race conditions

clunky, not fully parallel, but good performance!

rapidCFD approach!

std::linalg (C++26)
matrix_vector_product

ideal!

Different levels of integration

Level B: how to rewrite loops to avoid race conditions (RapidCFD approach):

$$A^T \cdot \psi = D \cdot \psi + L^T \cdot \psi + U^T \cdot \psi$$

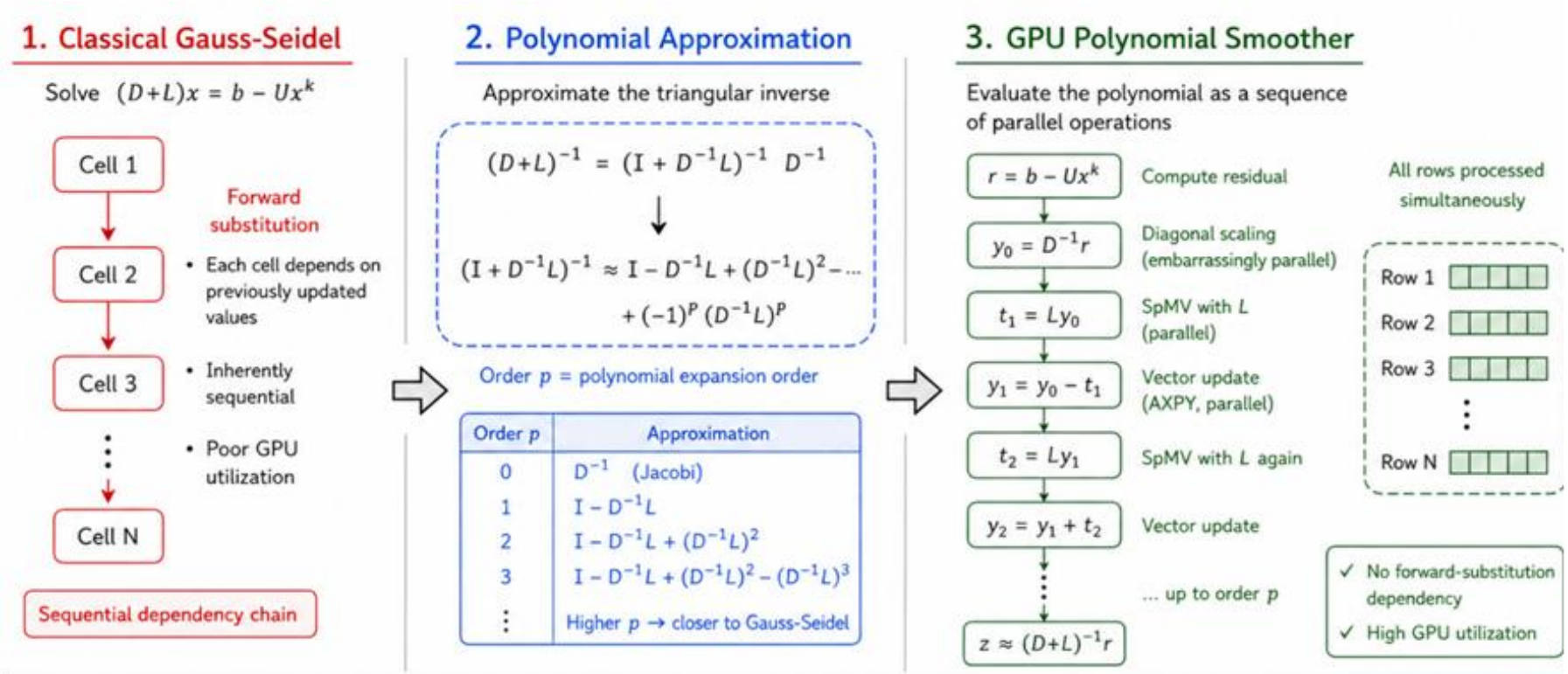
```
std::vector<Foam::label> indices(psi.size());
std::iota(indices.begin(), indices.end(), 0);

std::transform( std::execution::par_unseq,
                indices.begin(),
                indices.end(),
                Apsi.begin(),
                matrixMultiplyFunctor<3>
                (
                    tpsi,
                    diag(),
                    lower(),
                    upper(),
                    lduAddr().lowerAddr(),
                    lduAddr().upperAddr(),
                    lduAddr().ownerStartAddr(),
                    lduAddr().losortStartAddr(),
                    lduAddr().losortAddr()
                )

              );
```

```
for (Foam::label i = 0; i < nUnroll; i++)
{
    if (i < oSize)
    {
        Foam::label face = oStart + i;
        tmpSum[i] = upper[face] * psi[nei[face]];
    }
}

for (Foam::label i = 0; i < nUnroll; i++)
{
    if (i < nSize)
    {
        Foam::label face = nStart + i;
        face = losort[face];
        tmpSum[i + nUnroll] = lower[face] * psi[own[face]];
    }
}

for (Foam::label i = 0; i < 2 * nUnroll; i++)
{
    out += tmpSum[i];
}
```

```
Foam::scalar out = diag[id] * psi[id];
```

Different levels of integration

Level C: replace sequential algorithm with a fine grid parallel one



Key idea: Replace the inherently sequential triangular solve with a sequence of GPU-parallel sparse matrix–vector operations, providing a scalable approximation to Gauss–Seidel.

Reference:

•Berger-Vergiat et al., **Two-Stage Gauss–Seidel Preconditioners and Smoothers for Krylov Solvers on a GPU Cluster**, 2021.

•Thomas et al., **Neumann Series in GMRES and Algebraic Multigrid Smoothers**, 2021.

Offloading primitives:

Compilation

scalarField.H

```
BINARY_FUNCTION(scalar, scalar, scalar, pow)
```

```
#define BINARY_FUNCTION(ReturnType, Type1, Type2, Func) \
    BINARY_FUNCTION_TRANSFORM(ReturnType, Type1, Type2, Func) \
    BINARY_FUNCTION_INTERFACE(ReturnType, Type1, Type2, Func)
```

```
#define BINARY_FUNCTION_TRANSFORM(ReturnType, Type1, Type2, Func)            \
TEMPLATE                                                                     \
void Func(                                                                   \
    Field<ReturnType>& result,                                               \
    const UList<Type1>& f1,                                                  \
    const UList<Type2>& f2                                                   \
)                                                                            \
{                                                                            \
    std::transform                                                           \
    (                                                                        \
        std::execution::par_unseq,                                           \
        f1.begin(),                                                          \
        f1.end(),                                                            \
        f2.begin(),                                                          \
        result.begin(),                                                      \
        Func##BinaryFunctionFunctor<Type1,Type2,ReturnType>()                \
    );                                                                       \
}
```

Usage

k-ε model

```
omega_ = Cmu_ * pow(k_, 0.5) / epsilon_;
```

```
Foam::pow(Field<scalar>&, constField<scalar>&)
```

parallel execution!

```
std::transform(std::execution::par_unseq,
               k_.begin(), k_.end(),
               epsilon_.begin(),
               nuTilda_.begin(),
               powBinaryFunctionFunctor<scalar, scalar, scalar>());
```

Offloading primitives:

FieldM.H

```
GENERATE_BINARY_FUNCTION_FUNCTORS(pow)
```

```
template<class Type1,class Type2,class RType>
struct powBinaryFunctionFunctor
{
    RType operator()(const Type1& t1, const Type2& t2) const
    {
        return pow(t1, t2);  // will call your GPU-safe pow()
    }

    RType operator()(const std::tuple<Type1,Type2>& ts)
    {
        return pow(std::get<0>(ts), std::get<1>(ts));
    }
};
```

doubleFloat.H

```
#define MAXMIN(ReturnType, Type1, Type2) \
    MAXMIN(ReturnType, Type1, Type2) \
    __host__ __device__ inline ReturnType pow(const Type1 base, const Type2 expon) \
    { \
        return FOAM_POW_CALL(base, expon); \
    }
```

```
#define FOAM_POW_CALL(base, exp) \
    (::pow(static_cast<double>(base), static_cast<double>(exp)))
```

nvc++

__nv_pow

clang++

```
#if defined(__HIP_DEVICE_COMPILE__)
    #define FOAM_POW_CALL(base, exp) \
        __ocml_pow_f64(static_cast<double>(base), static_cast<double>(exp))
#else
    #define FOAM_POW_CALL(base, exp) \
        (::pow(static_cast<double>(base), static_cast<double>(exp)))
#endif
```

Generally we found that: **Clang's offload lowering rules are primarily syntactic**, while the NVIDIA HPC C++ compiler (**nvc++**) is generally described as having more **semantic** offloading capabilities!

Current performance

We tested on 4 different main test case (https://develop.openfoam.com/committees/hpc/-/tree/update-README-exaFOAM):

Test case	Resolution	solver	preconditioner	matrix solver
cavity3D	1M, 8M, 64M	icoFoam	diagonal	PCG, GAMG
conical diffuser	3M	simpleFoam	twoStepGaussSeidel	GAMG
HPC motorbike	Small, Medium, Large	simpleFoam	twoStepGaussSeidel	PCG, GAMG
drivAer	5M	simpleFoam	twoStepGaussSeidel	GAMG

on the following hardware (w/o UMPIRE):

- 32-core: a 32-core Intel Xeon Gold 6448Y CPU
- H100: 1-core from Intel Xeon Gold 6448Y CPU + 1 NVIDIA H100

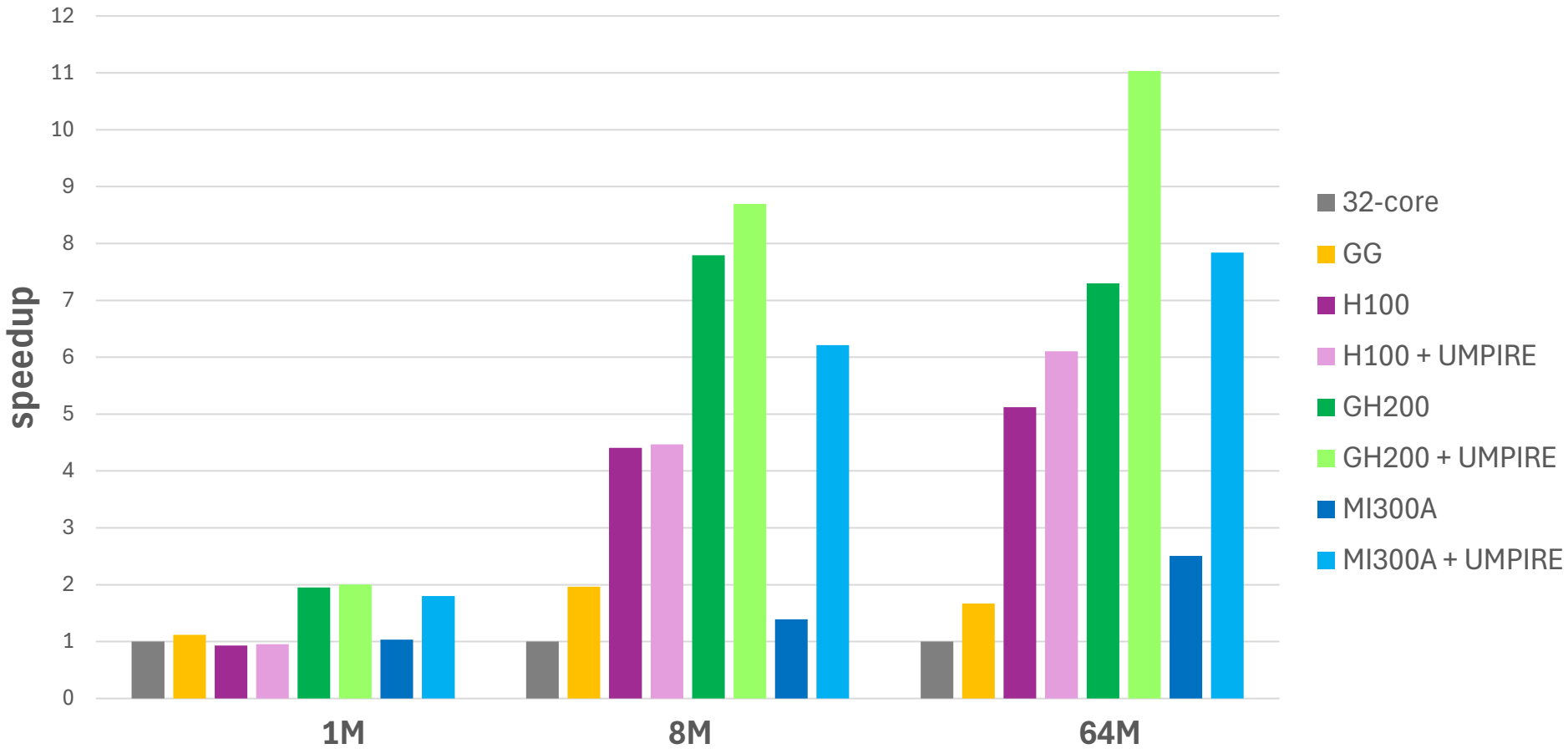
} STFC

- GG: Grace-Grace 144-coreARMNeoverse-V2
- GH200: NVIDIA Grace Hopper

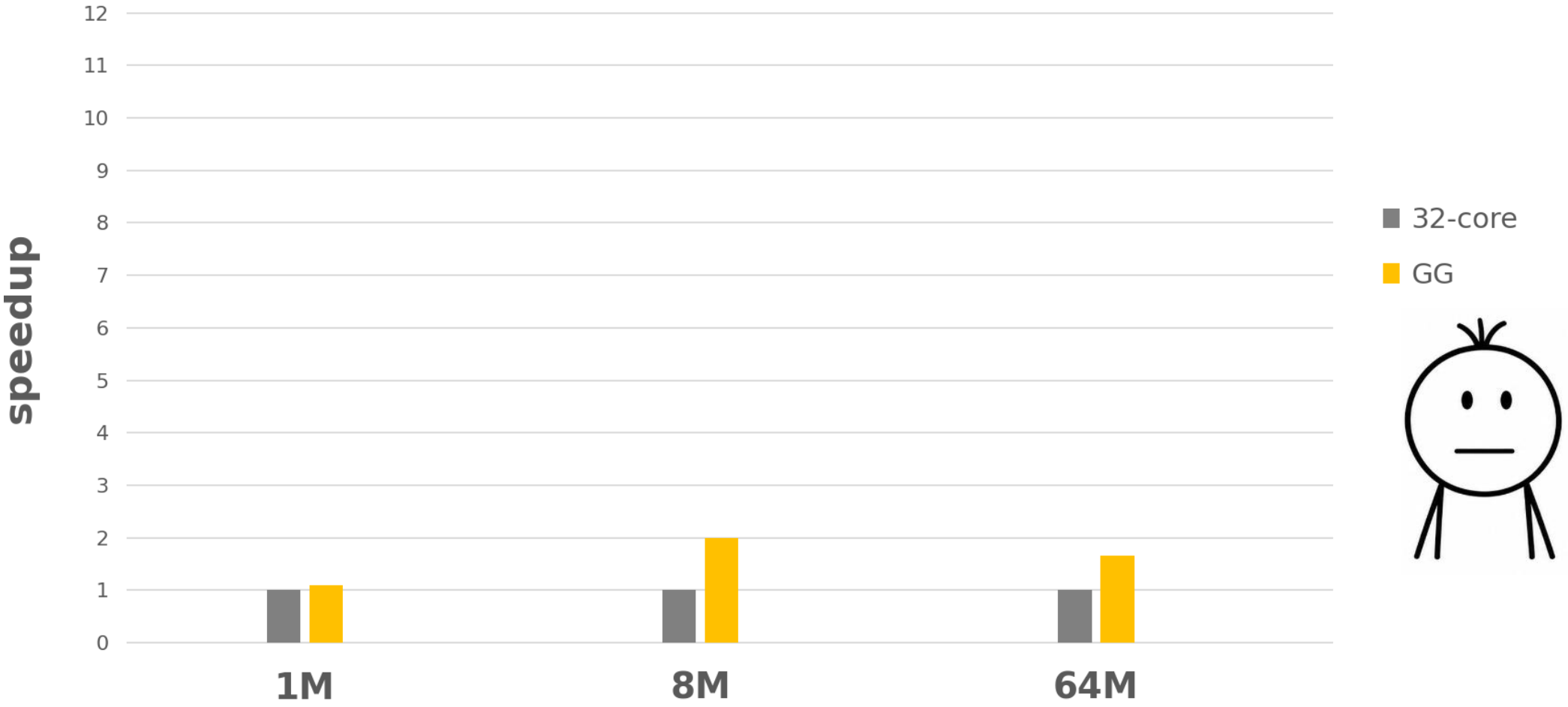
} from the NVIDIA "Thea" MGX Evaluation testbed.

- MI300A: 1 AMD MI300A Durham COSMA supercomputer

cavity3D (PCG)



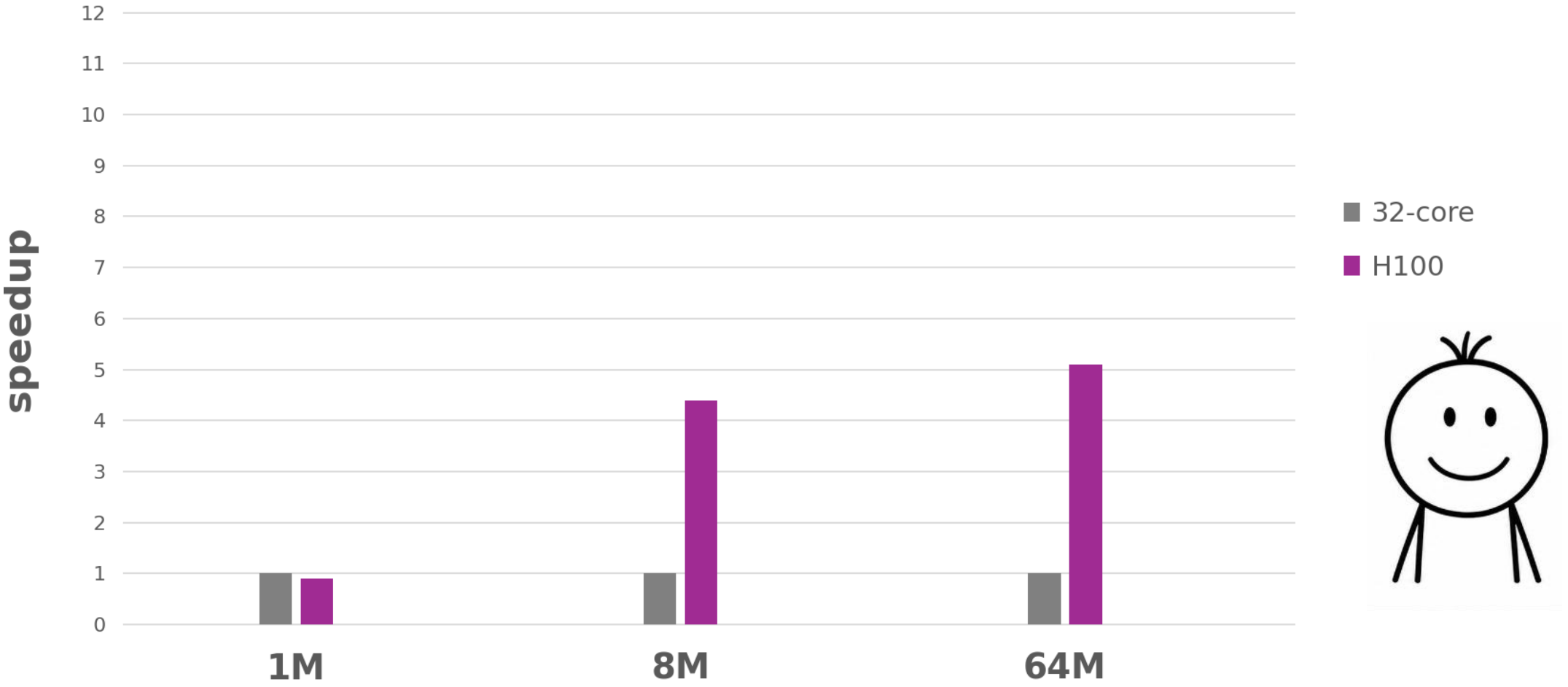
cavity3D (PCG)



Compiler flags

-stdpar=multicore

cavity3D (PCG)

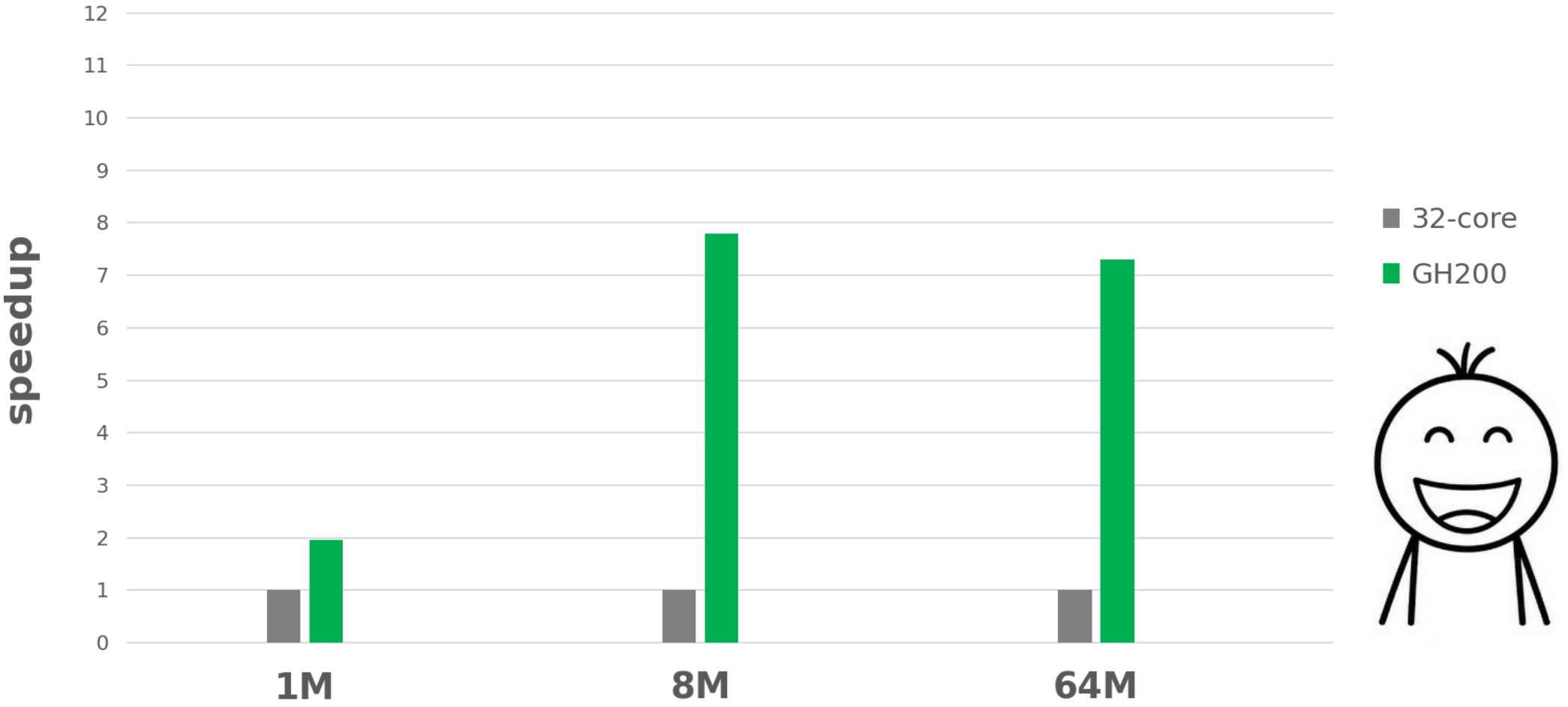


Compiler flags

-stdpar=gpu -gpu:cc90;mem:managed

nvidia-smi -q | grep -i "Addressing Mode“ = HMM

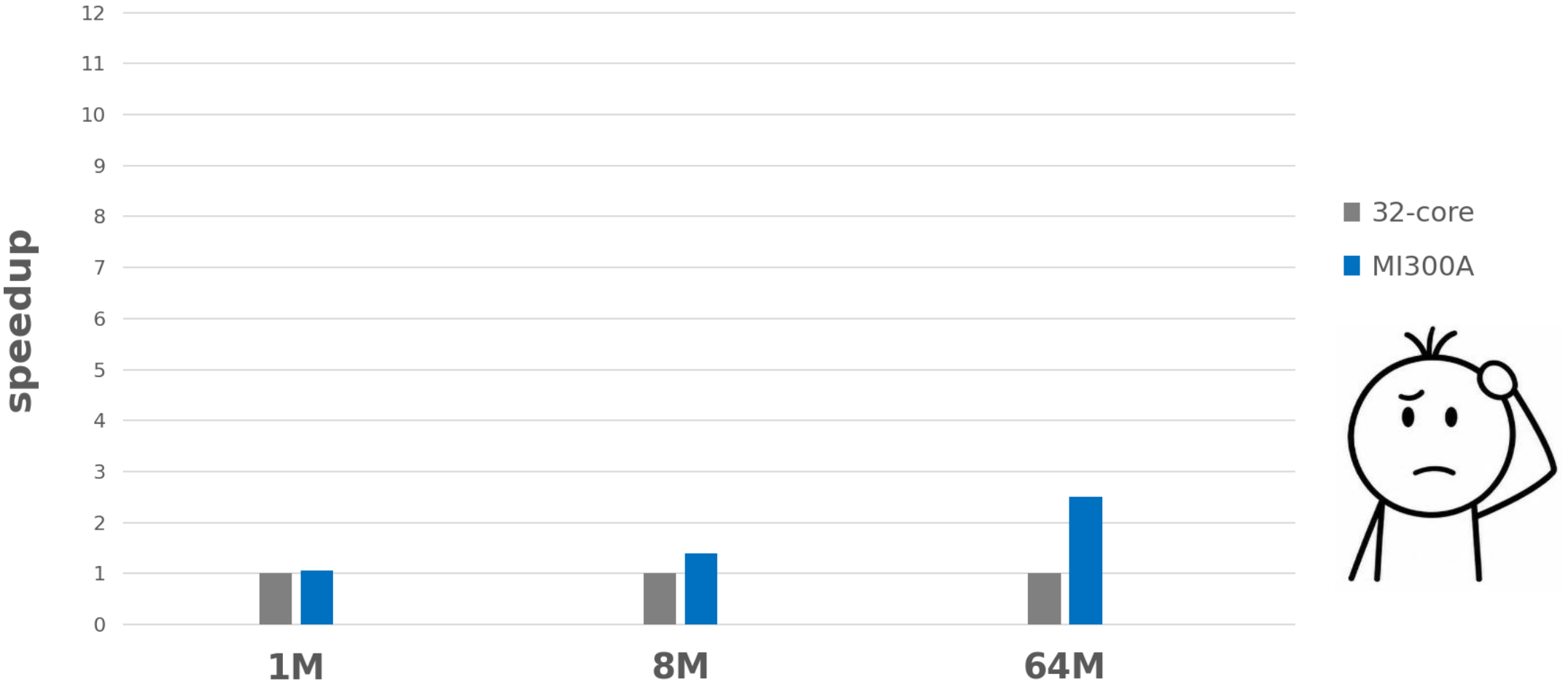
cavity3D (PCG)



Compiler flags

-stdpar=gpu -gpu:cc90,mem:unified:managed

cavity3D (PCG)



Compiler flags

--hipstdpar --offload-arch=gfx942
export HSA_XNACK=1

The magic on UMPIRE!

UMPIRE is a cross-platform memory management library that abstracts CPU and GPU memory allocation, movement, and tracking, enabling portable high-performance applications.

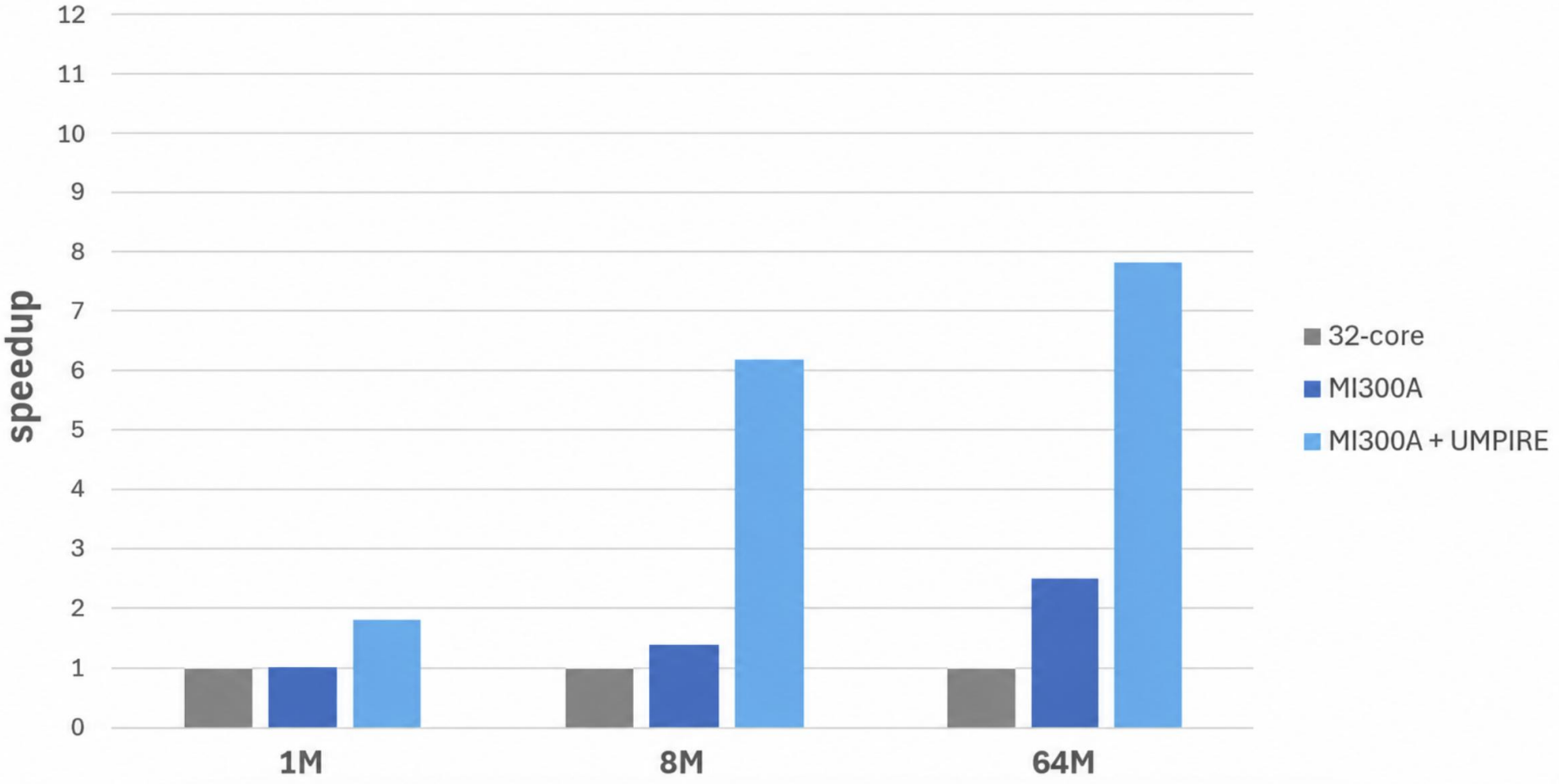
Lawrence Livermore National Laboratory

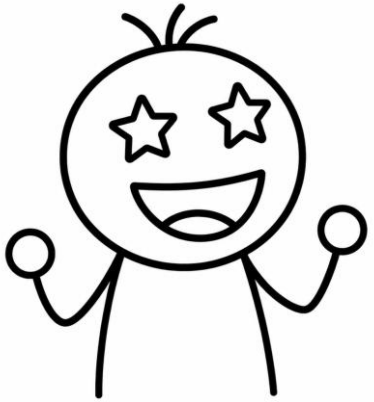
In the latest version of OpenFOAM you will find instruction to use it:

makeUMPIRE

```
auto& rm = umpire::ResourceManager::getInstance();

auto host   = rm.getAllocator("HOST");
auto device = rm.getAllocator("DEVICE");

double* h = static_cast<double*>(host.allocate(N*sizeof(double)));
double* d = static_cast<double*>(device.allocate(N*sizeof(double)));

rm.copy(d, h, N*sizeof(double));   // Host → Device
...
rm.copy(h, d, N*sizeof(double));   // Device → Host

host.deallocate(h);
device.deallocate(d);
```

cavity3D (PCG)

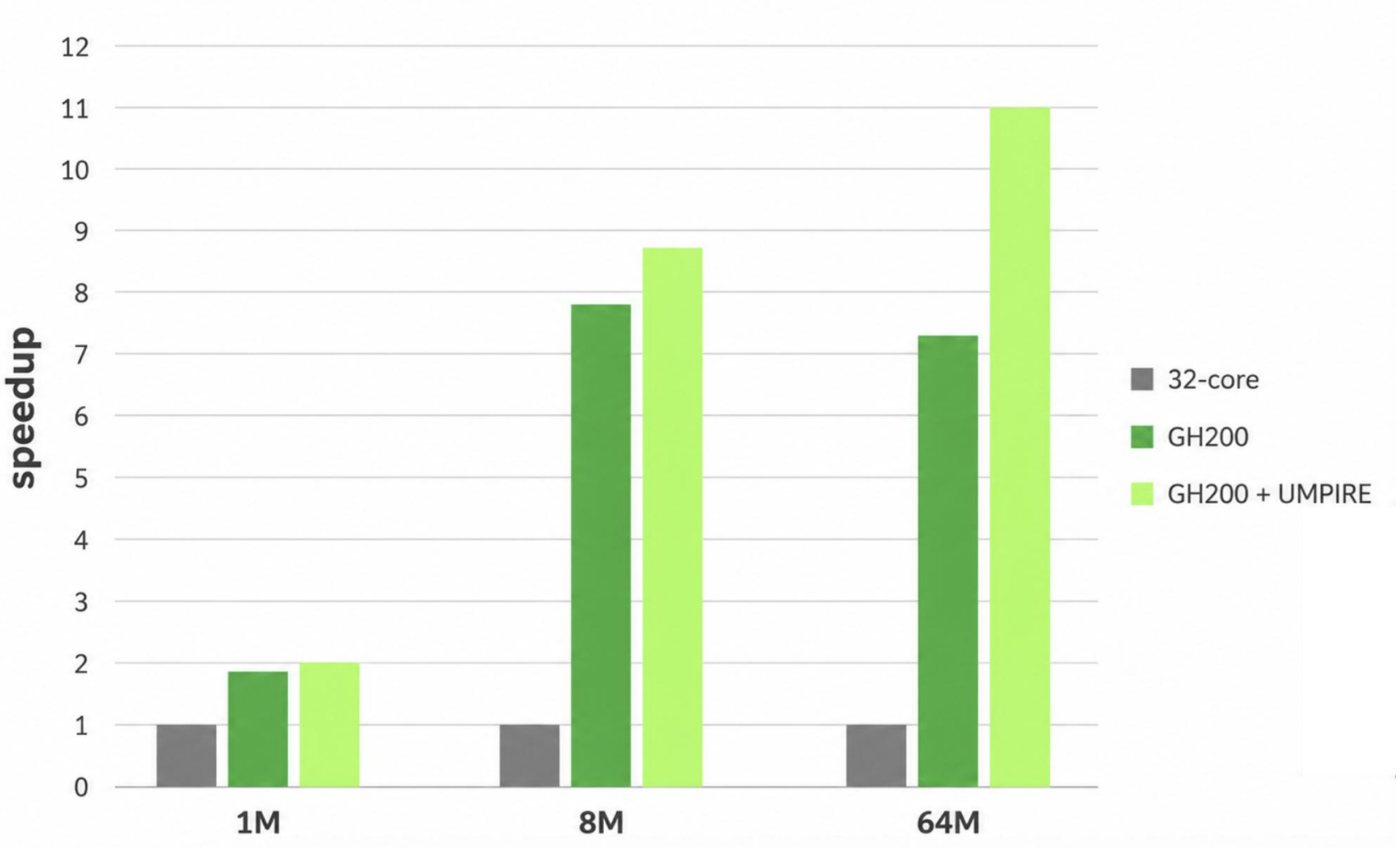




Compiler flags

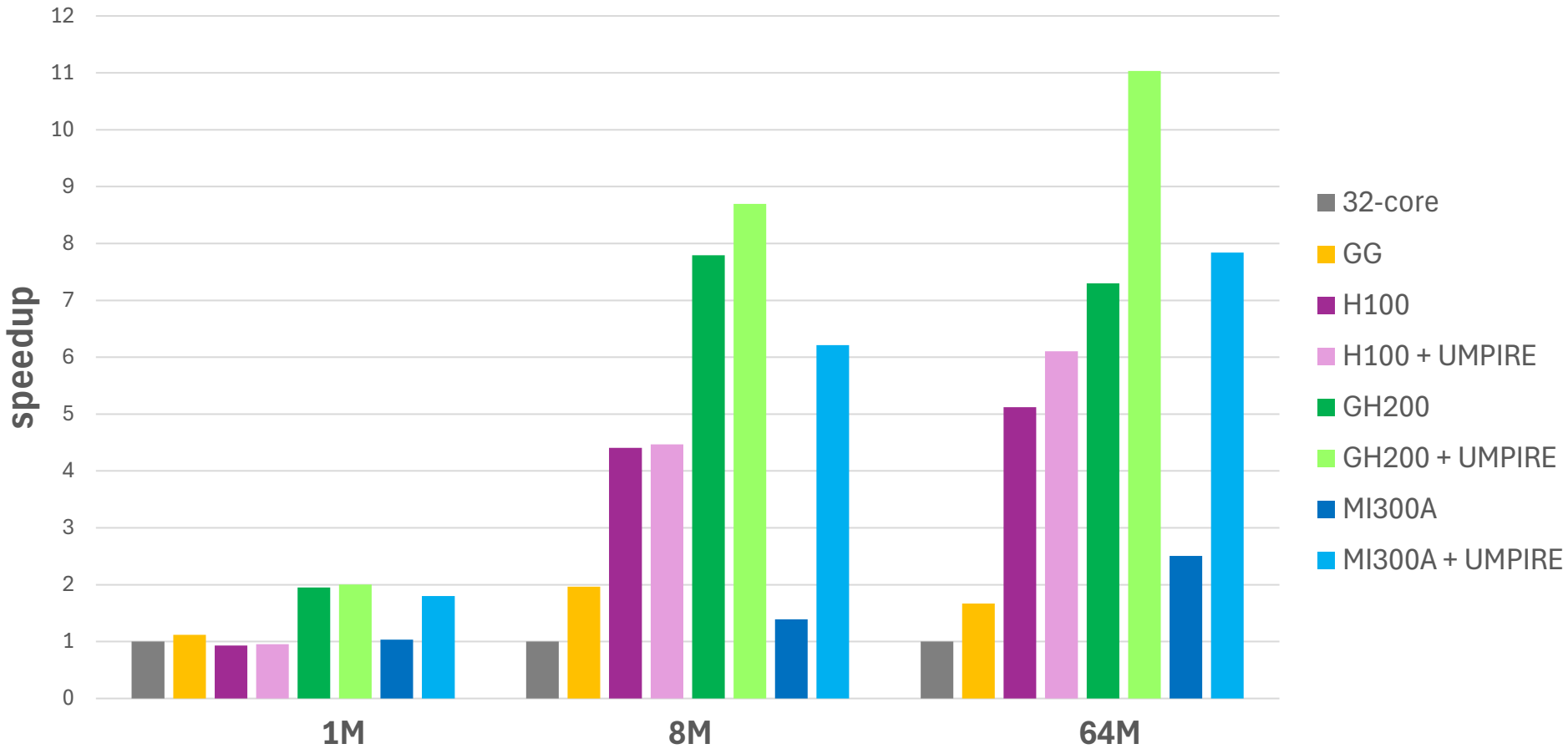
export FOAM_MEMORY_POOL="system; size=80240; incr=1024"

cavity3D (PCG)

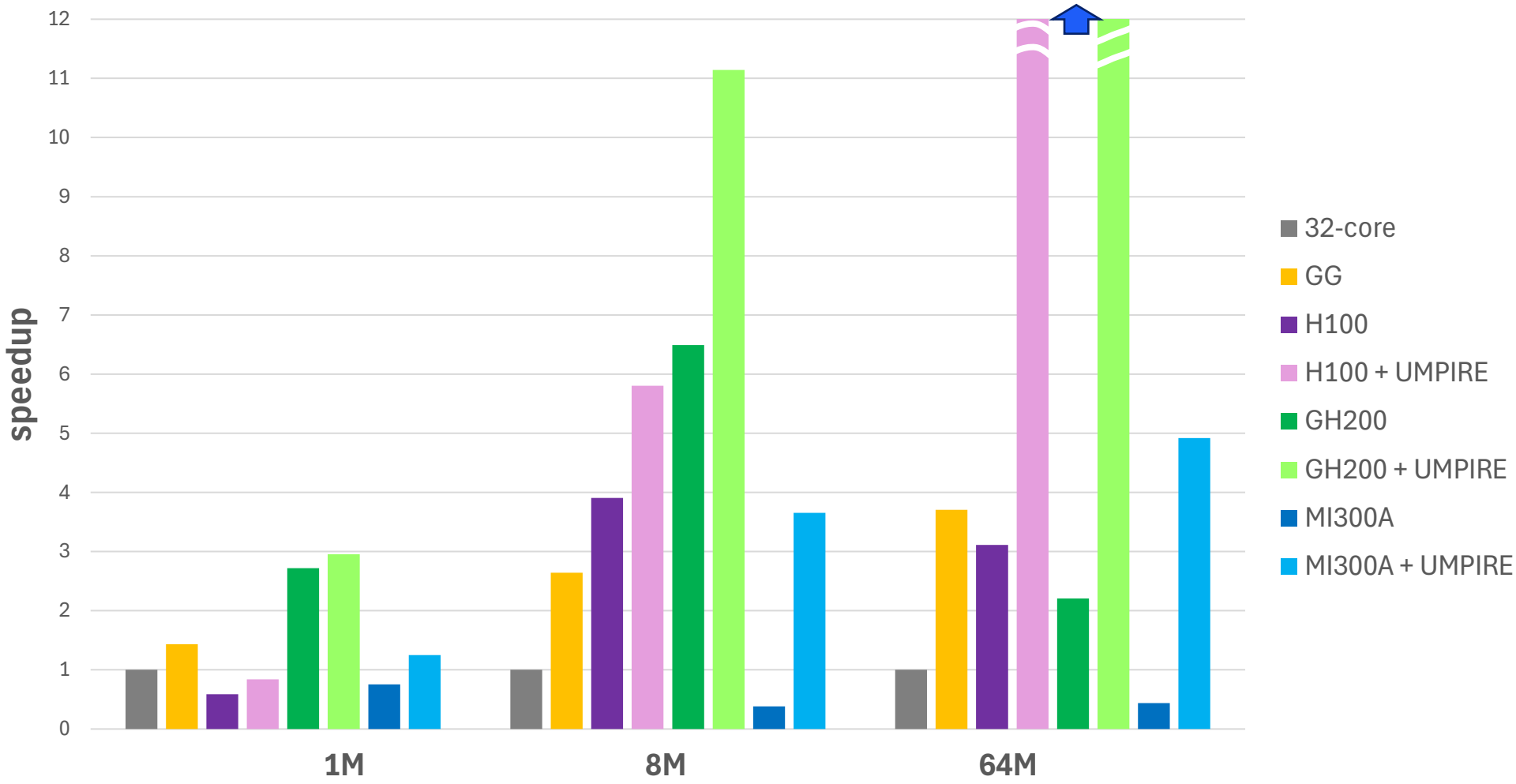


Compiler flags export FOAM_MEMORY_POOL="managed; size=80240; incr=1024"

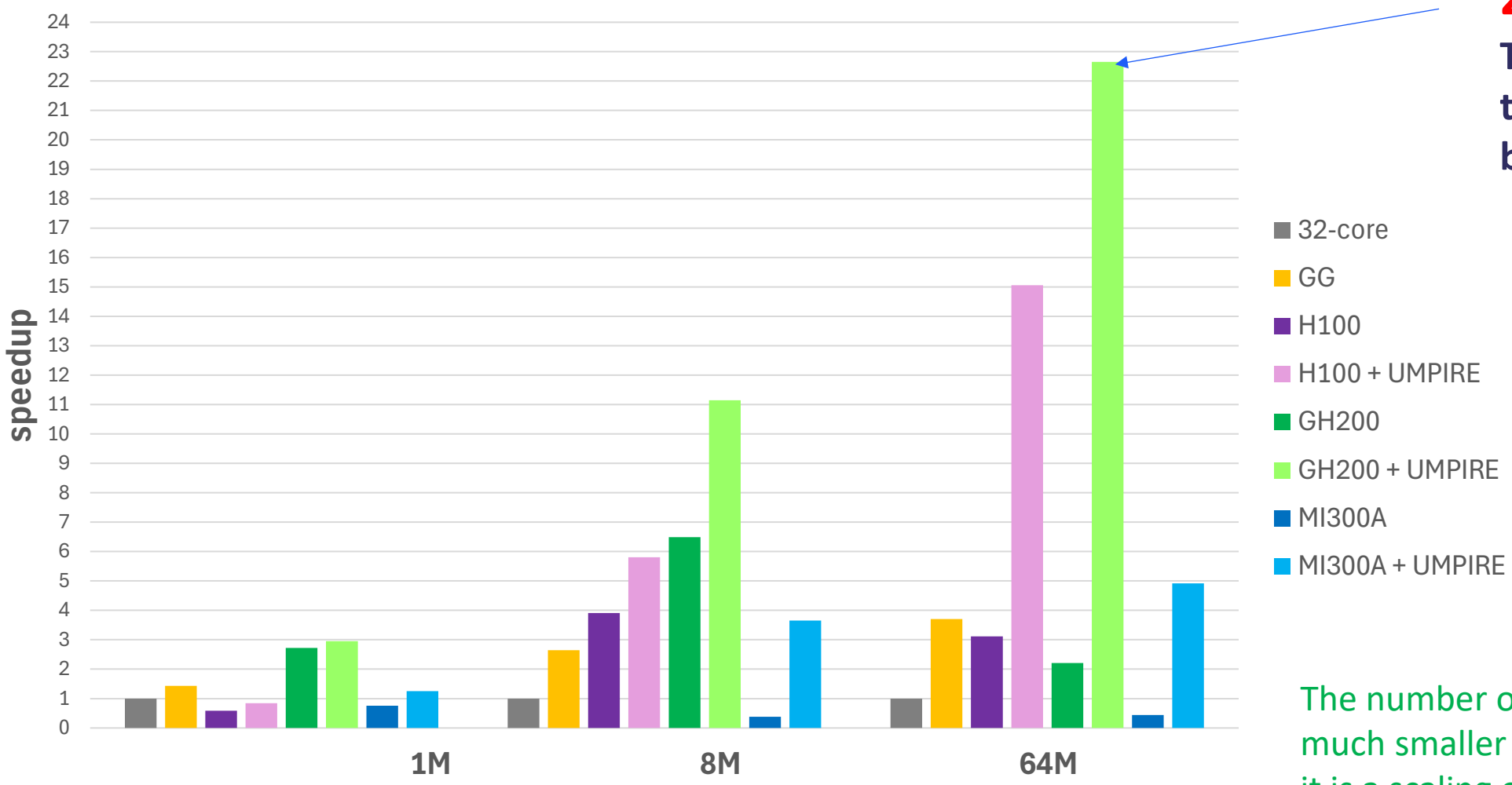
cavity3D (PCG)



cavity3D (sGS-GAMG)

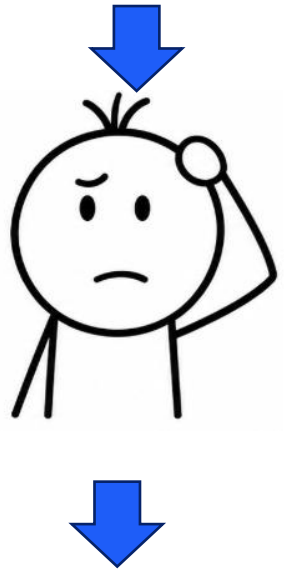


cavity3D (sGS-GAMG)



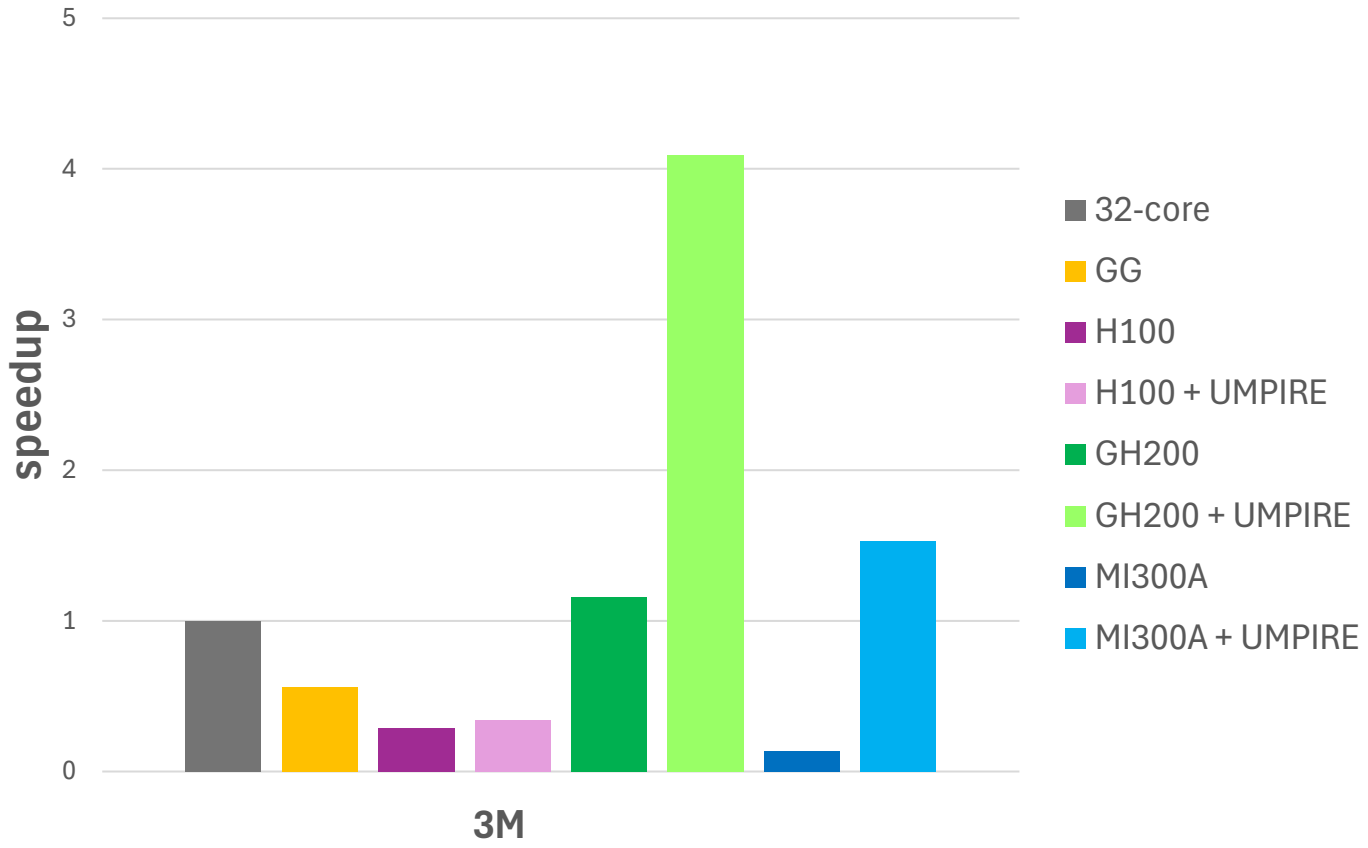
22x!

This is well beyond the memory bandwidth ratio…

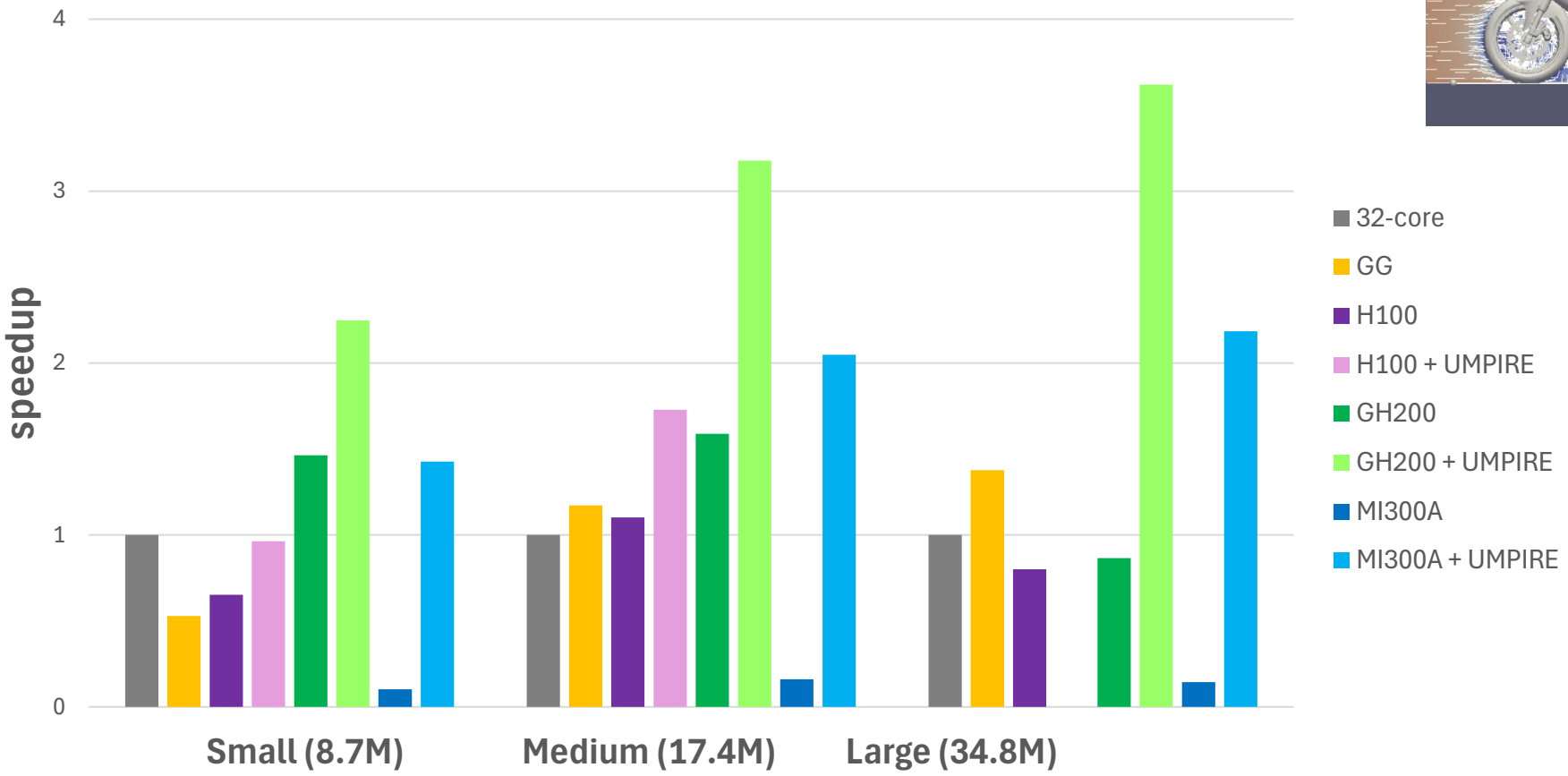


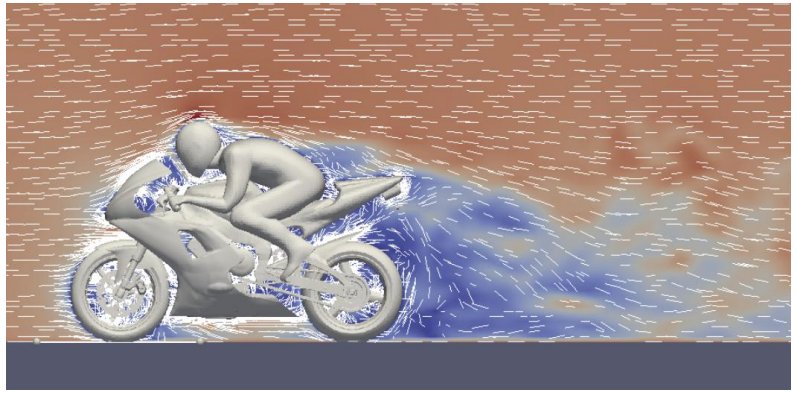
The number of iteration on 1 GPU is much smaller (same as 1 MPI task). So it is a scaling of GAMG issue with MPI!

Conical diffuser

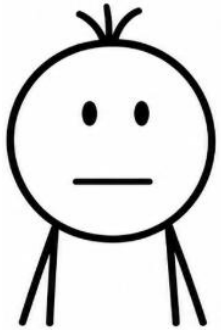


HPC motorbike (PCG)

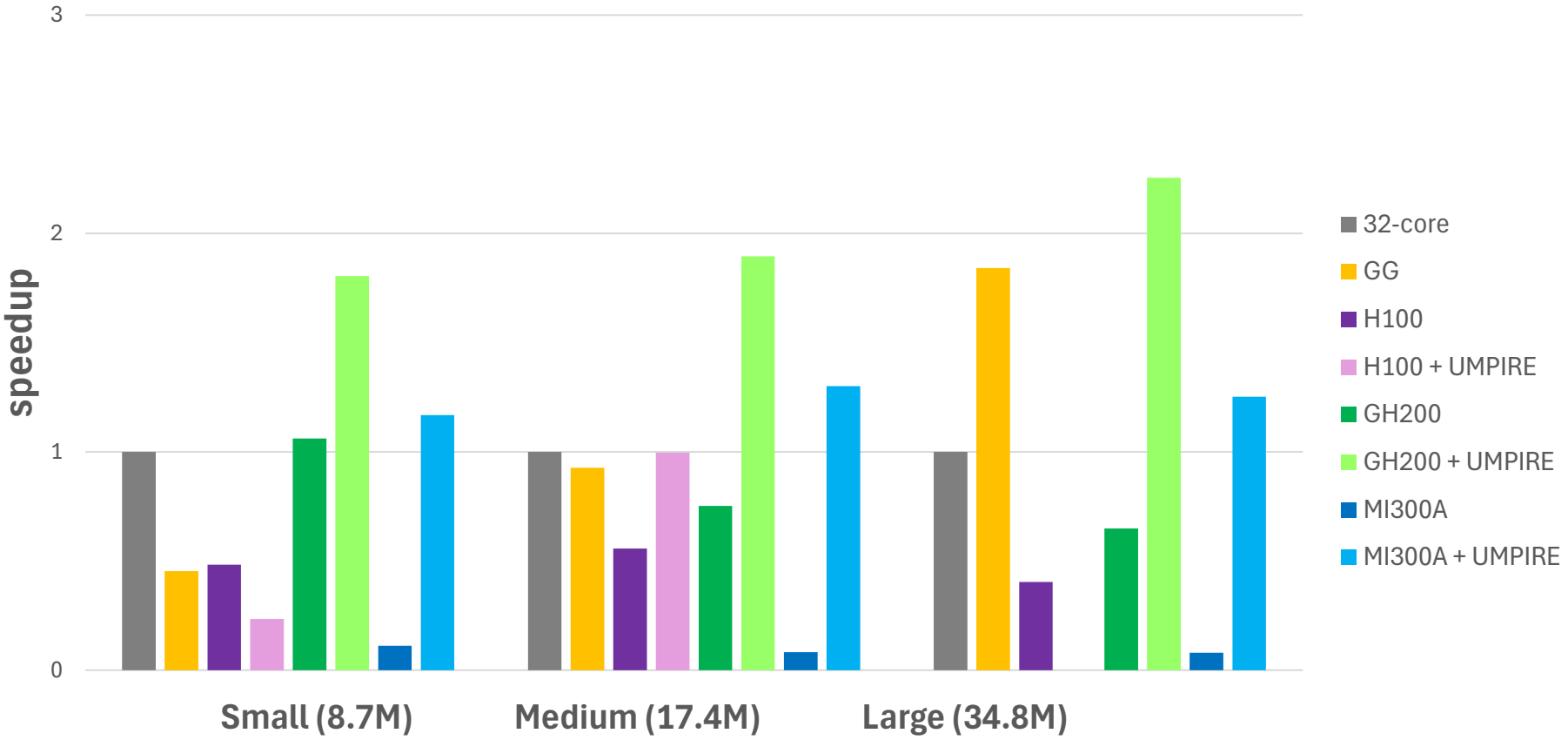




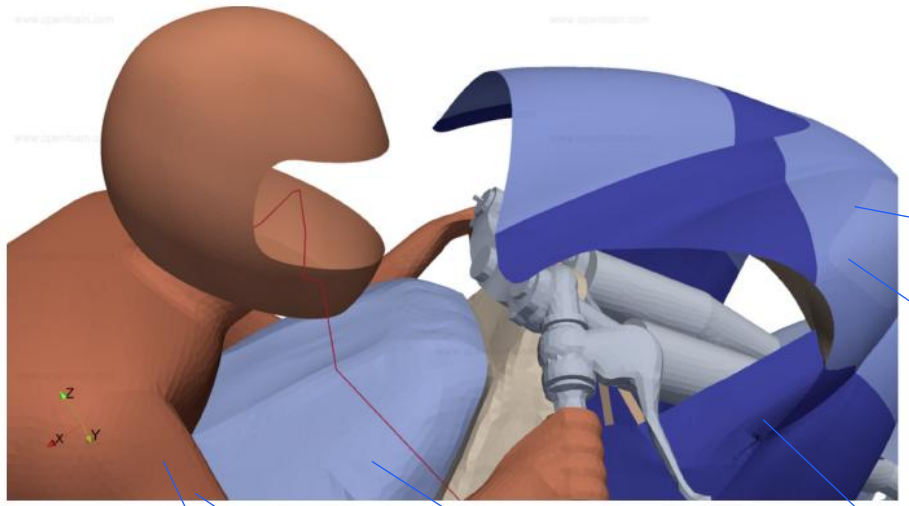
OpenFOAM tutorial



HPC motorbike (sGS-GAMG)



HPC motorbike (sGS-GAMG)



kernel 1

kernel 2

kernel 3

kernel 4

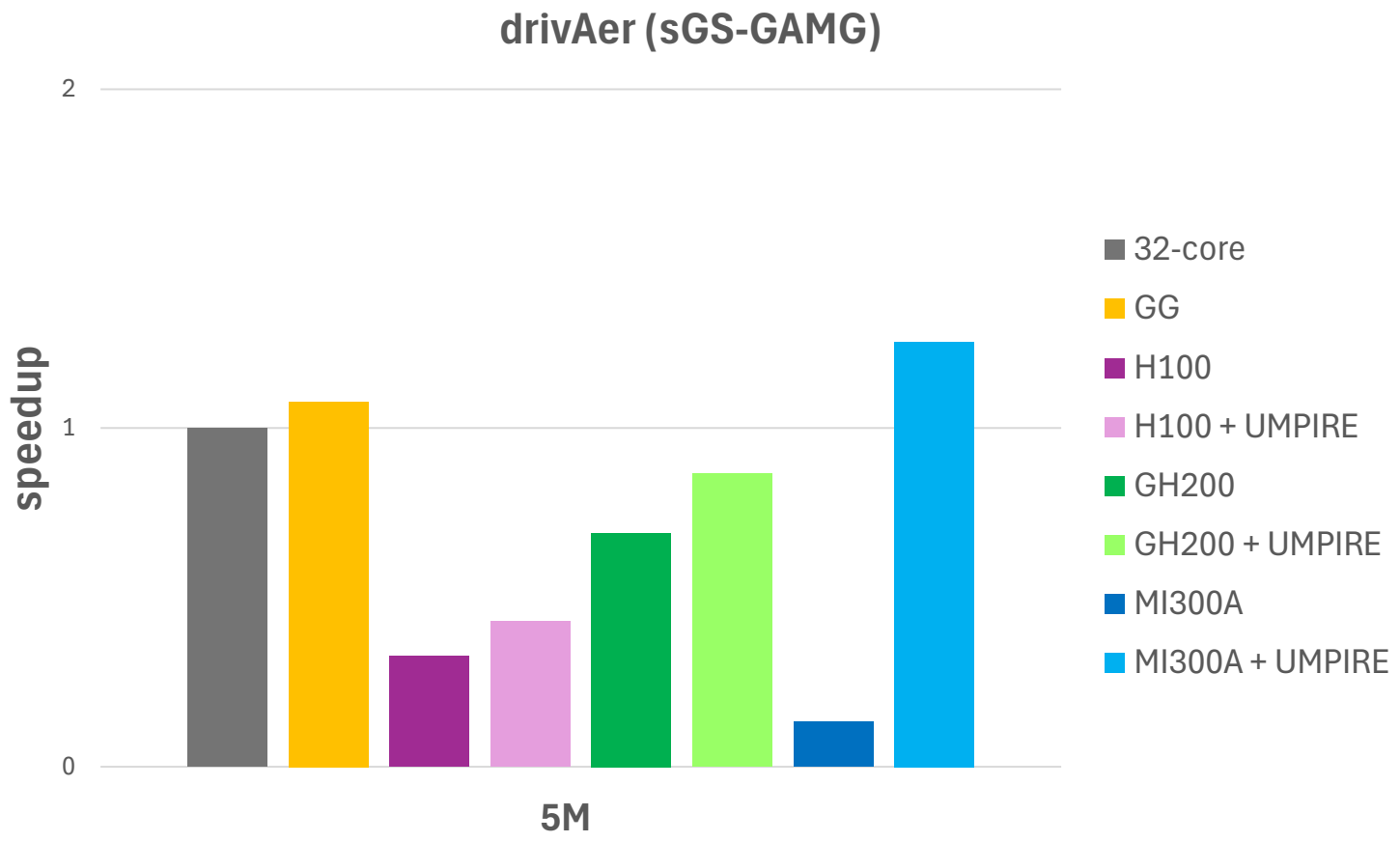
kernel 5

kernel …

A lot of small kernels are generated and this impacts detrimentally on the performance

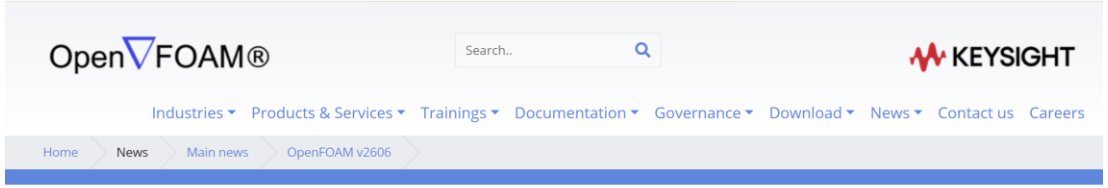
Group patches (OpenCFD)

drivAer (sGS-GAMG)



Still thinking…

How to use it



v2606: New and improved infrastructure

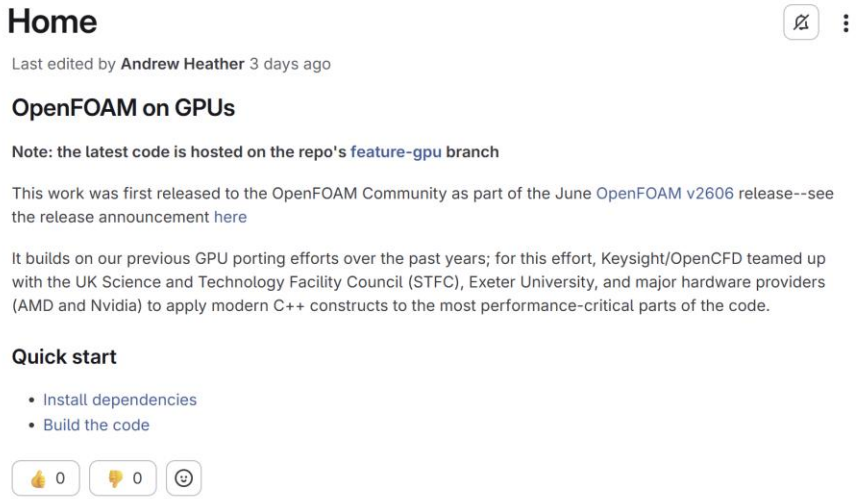
Community contribution: GPU

TOP

This is the first release to support GPU offloading. It uses the C++17/20 `std::execution` policy to automatically run loops in parallel across multiple execution units, which may be GPU devices or cores sharing memory.

https://www.openfoam.com/news/main-news/openfoam-v2606/infrastructure#gpu

Follow the instructions on Wiki!



Home

Last edited by **Andrew Heather** 3 days ago

OpenFOAM on GPUs

Note: the latest code is hosted on the repo's feature-gpu branch

This work was first released to the OpenFOAM Community as part of the June OpenFOAM v2606 release--see the release announcement here

It builds on our previous GPU porting efforts over the past years; for this effort, Keysight/OpenCFD teamed up with the UK Science and Technology Facility Council (STFC), Exeter University, and major hardware providers (AMD and Nvidia) to apply modern C++ constructs to the most performance-critical parts of the code.

Quick start

- Install dependencies
- Build the code

0 0

https://gitlab.com/openfoam/gpu/openfoam-ecse/-/wikis/home

Contacts: Jony.Castagna@stfc.ac.uk
Mayank.kumar@stfc.ac.uk

CPU/GPU compilation

1 source code!

Use __FOAM_OFFLOAD__ pre-processor flag to support offloading. If not, it will compile for CPU (parallel MPI) execution!

for Nvidia GPU, compile using:
WM_COMPILER=Nvidia-gpu

nvc++ -std=c++20 -stdpar=gpu -gpu=mem:unified:managedalloc -DFOAM_OFFLOAD → Nvidia GPU
or
nvc++ -std=c++20 -stdpar=multicore -DFOAM_OFFLOAD → Any CPU (Intel, AMD , ARM)

for AMD GPU, compile using:
WM_COMPILER=Amd-gpu

amdclang++ -std=c++20 --hipstdpar -DFOAM_OFFLOAD → AMD GPU
or
amdclang++ -std=c++20 --hipstdpar -stdpar=multicore -DFOAM_OFFLOAD → Any CPU (Intel, AMD , ARM)

Remember to

1. enable HMM on heterogenous NVIDIA GPU

```
nvidia-smi -q | grep -i "Addressing Mode"
    Addressing Mode                       : HMM
    Addressing Mode                       : HMM
```

2. compile UMPIRE according to your architecture

3. compile openFoam (feature-gpu branch) according to your architecture

```
=======================================
Compile OpenFOAM libraries
=======================================
    ln: OpenFOAM/lnInclude
    ln: OSspecific/POSIX/lnInclude
    found umpire -- enabling memory pool interface
    umpire libs: -lumpire -lfmt -lcamp
wmake libo (POSIX)
wmake -no-openmp dummy (mpi=SYSTEMOPENMPI)
```

4. modify preconditioner (diagonal or twoStepGaussSeidel)

```
solvers
{
    p
    {
        solver          GAMG;
        smoother        twoStageGaussSeidel;
        tolerance       1e-5;
        relTol          0.05;
        maxIter         3000;
    }
```

5. export FOAM_MEMORY_POOL=… according to your architecture

6. execute as single core (icoFoam, not mpirun –np …)

Still to do… a lot!

- Improve performance real industrial cases
- Merge into the CPU version (2612 release!)
- Multi-GPU
- Port other solver, turbulence models, etc
- Help is welcome!

Thank you!

(and please remember: Mayank Kumar is the main developer!!)