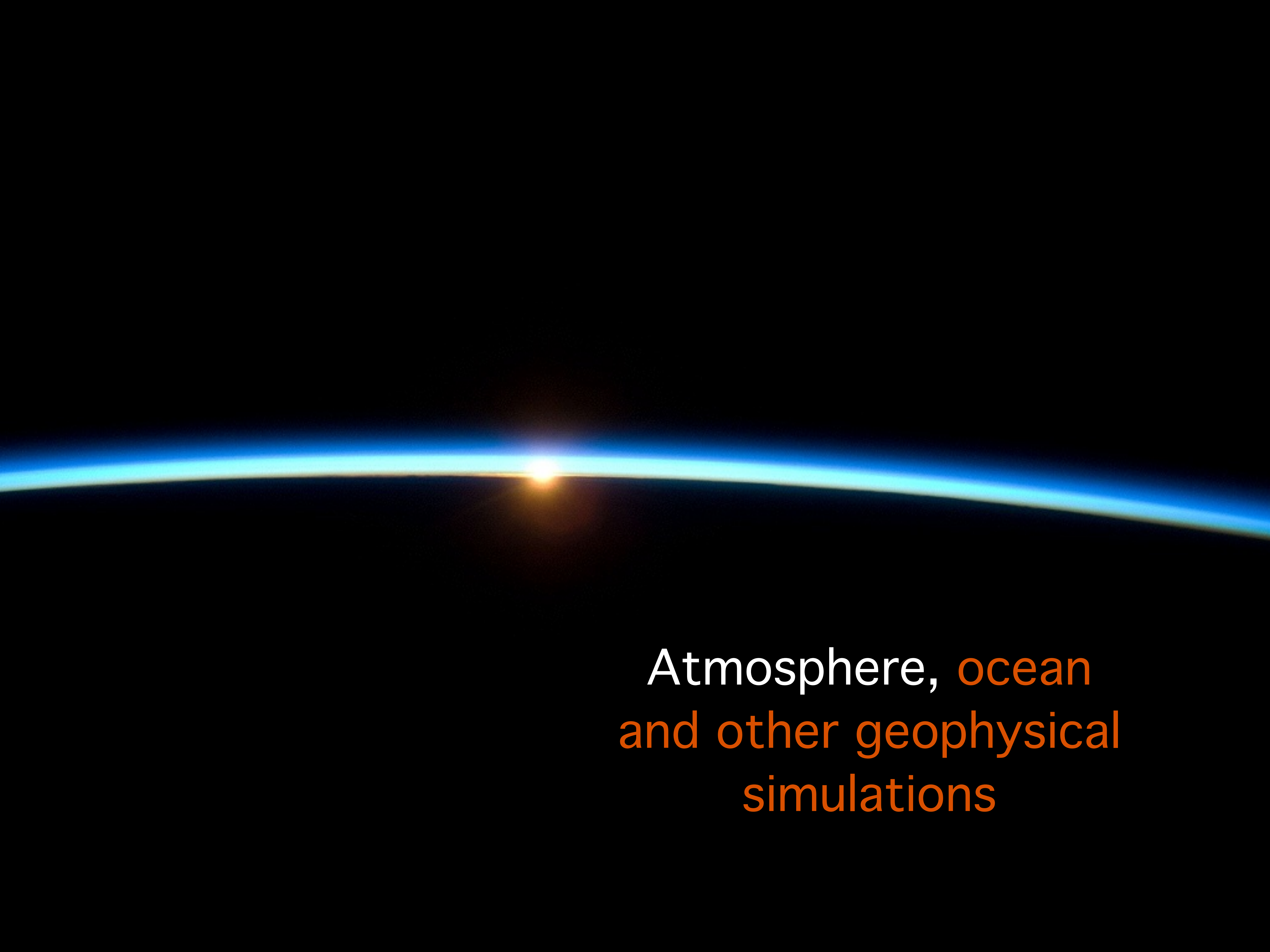
Firedrake: Extruded Meshes and Outer-Product Elements

Speaker 1:
Gheorghe-Teodor (Doru) Bercea

Speaker 2:
Andrew T.T. McRae

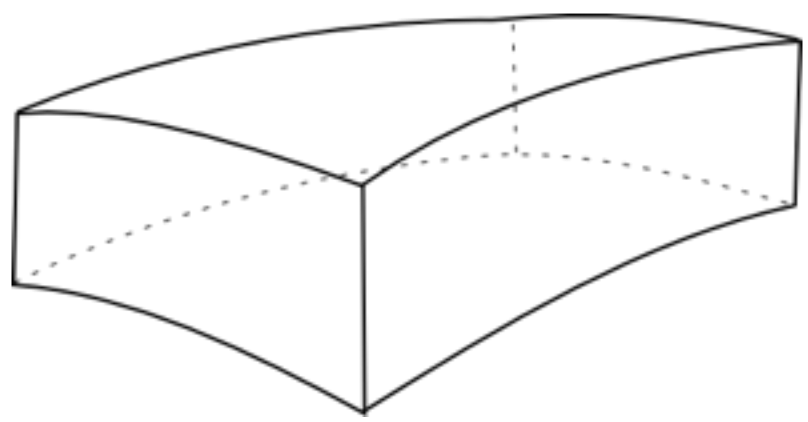
Florian Rathgeber, Lawrence Mitchell, Fabio Luporini,
Colin J. Cotter, David A. Ham, Paul H. J. Kelly

Imperial College London



Atmosphere, ocean and other geophysical simulations

Numerical considerations for atmosphere and ocean simulations

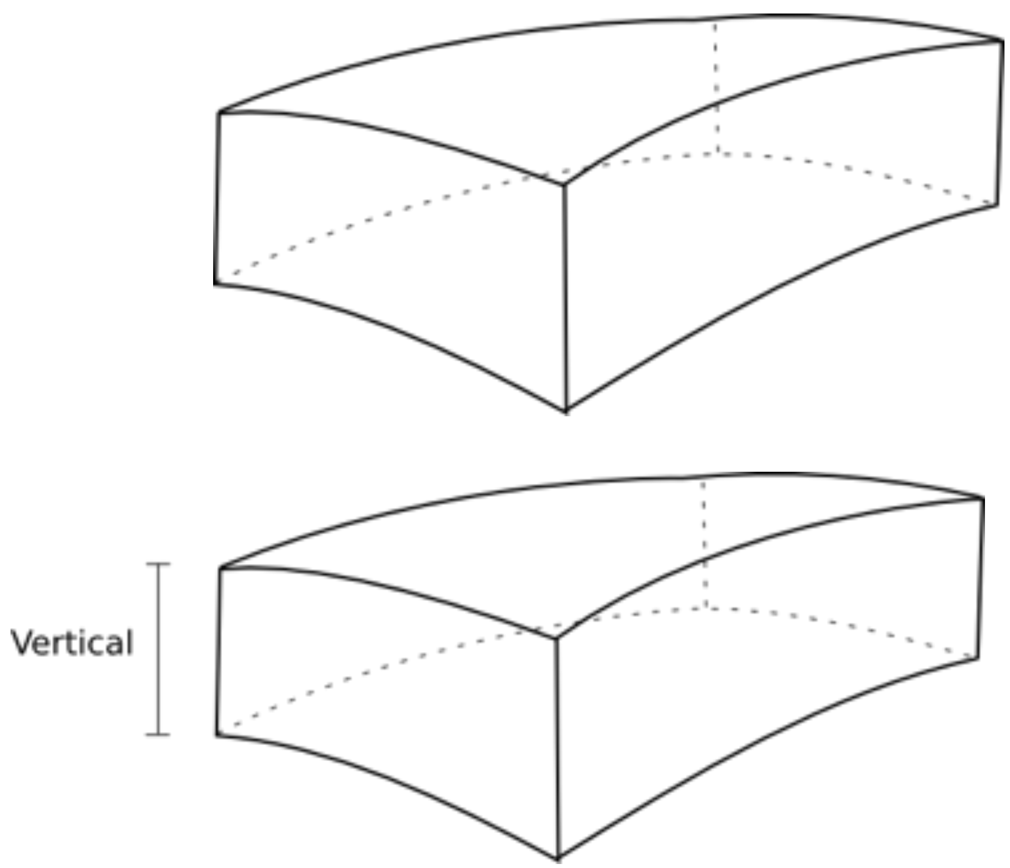


In these very thin domains, there is a large scale separation between the physics and numerics in the horizontal and vertical directions.

This makes vertically aligned meshes numerically advantageous.

We will also show that they have performance advantages.

Numerical considerations for atmosphere and ocean simulations

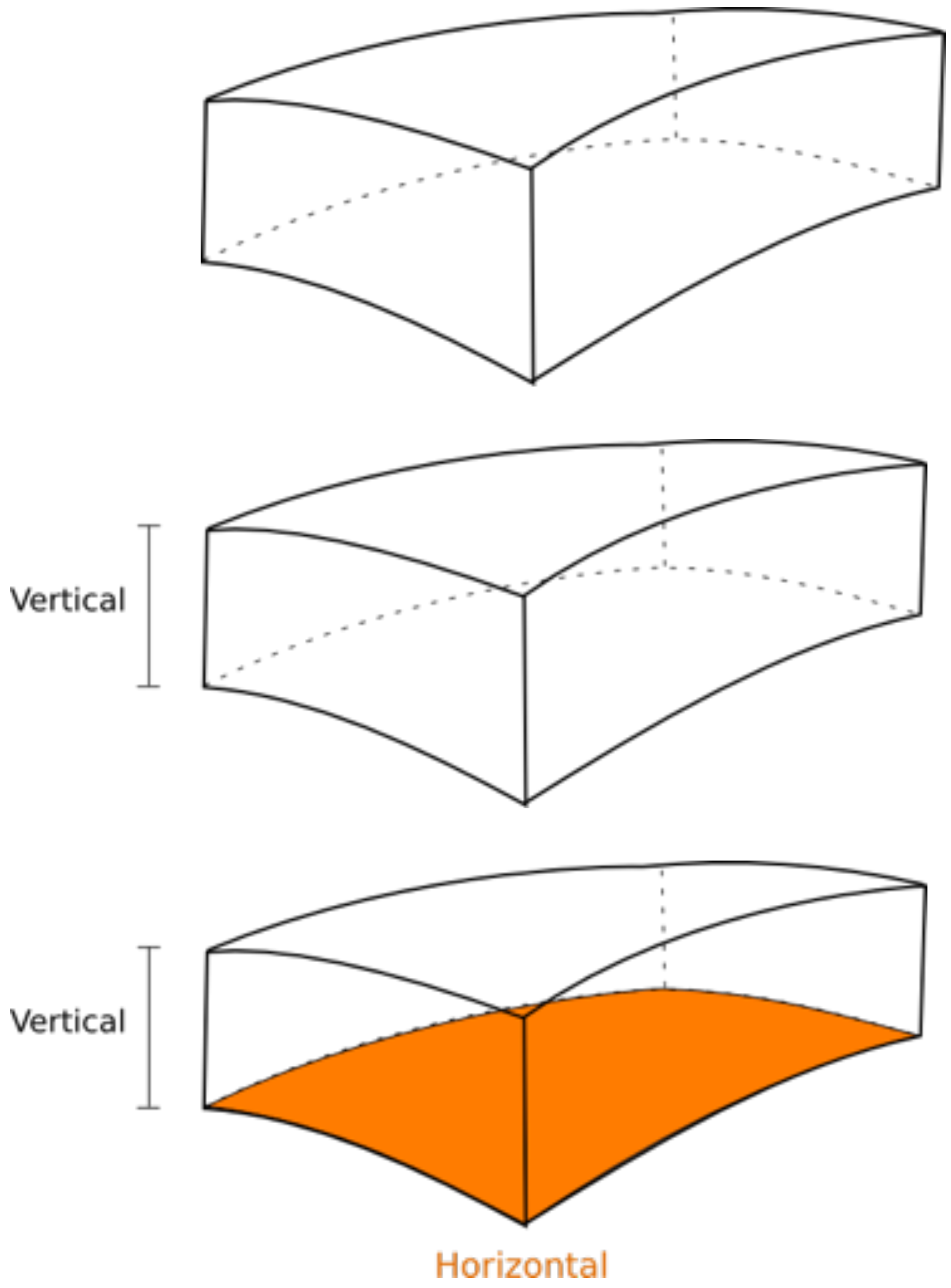


In these very thin domains, there is a large scale separation between the physics and numerics in the horizontal and vertical directions.

This makes vertically aligned meshes numerically advantageous.

We will also show that they have performance advantages.

Numerical considerations for atmosphere and ocean simulations



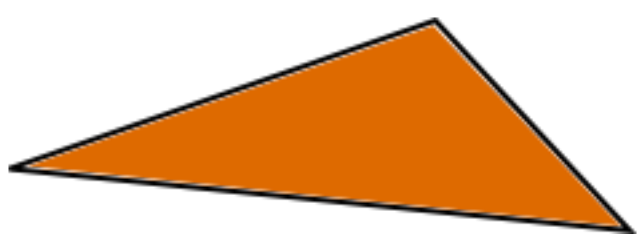
In these very thin domains, there is a large scale separation between the physics and numerics in the <u>horizontal</u> and <u>vertical</u> directions.

This makes vertically aligned meshes numerically advantageous.

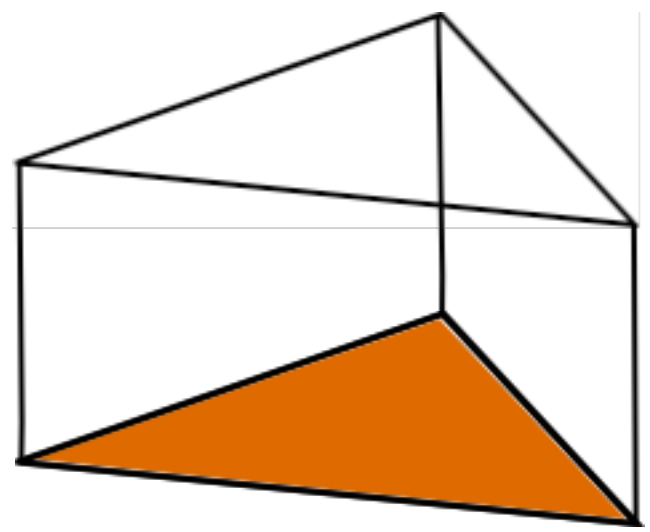
We will also show that they have performance advantages.

Vertical Alignment

Also good for the performance side!

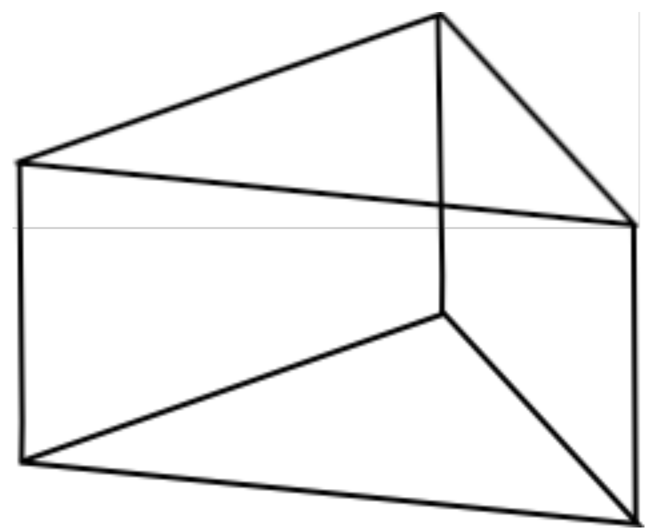


Vertical Alignment



Also good for the performance side!

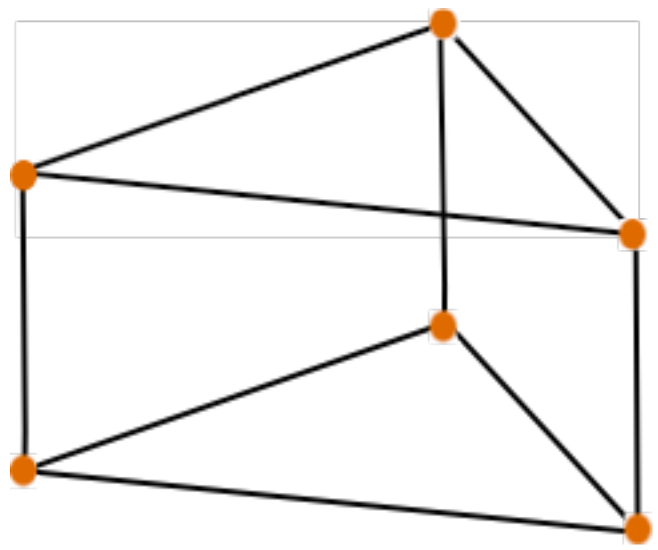
Vertical Alignment



Also good for the performance side!

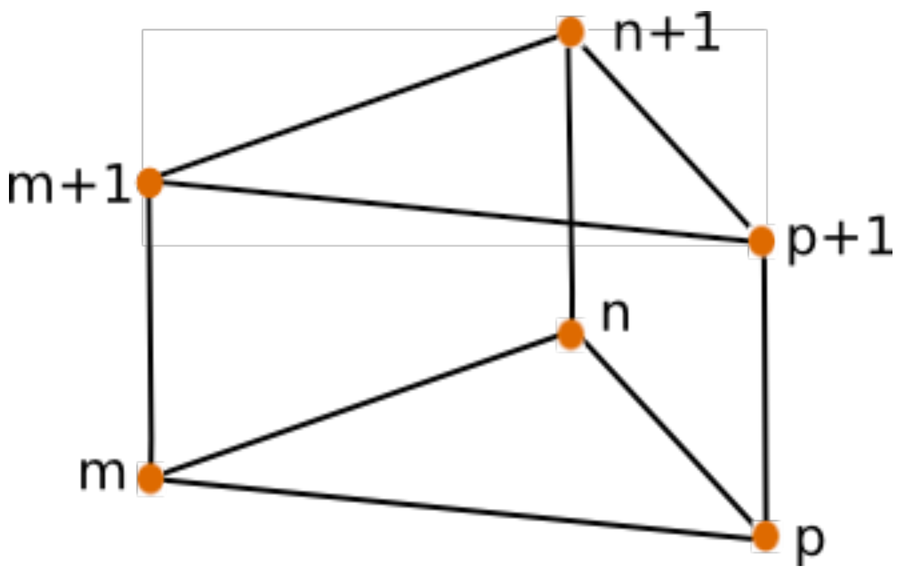
Vertical Alignment

Also good for the performance side!

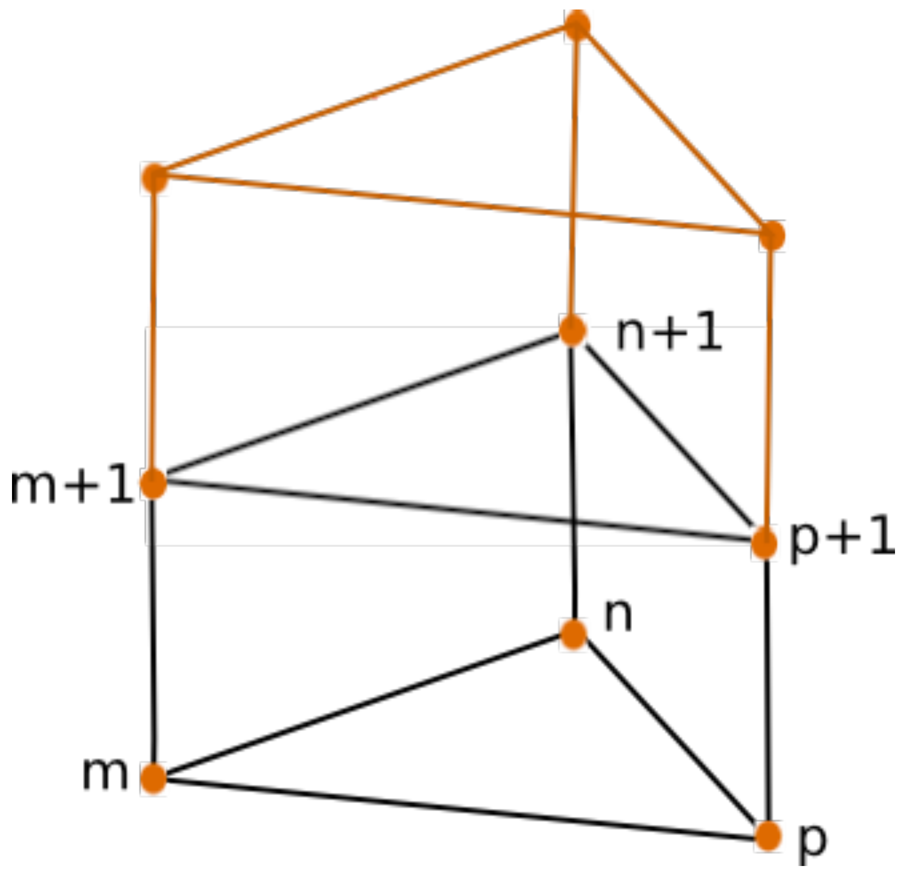


Vertical Alignment

Also good for the performance side!

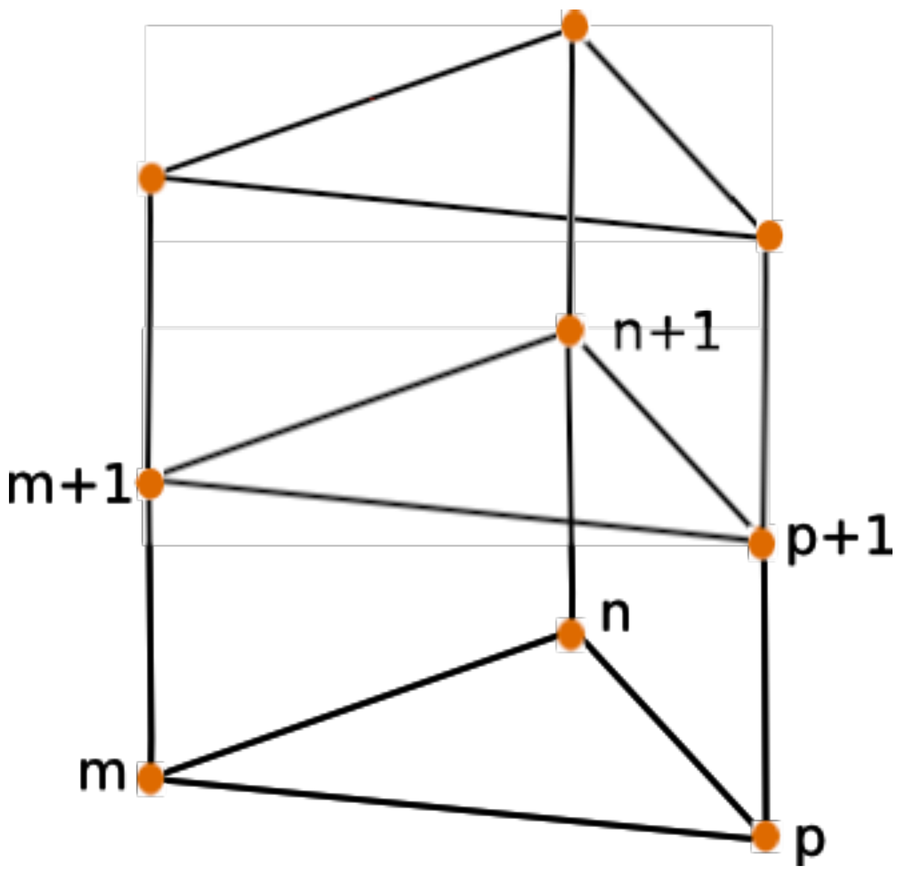


Vertical Alignment



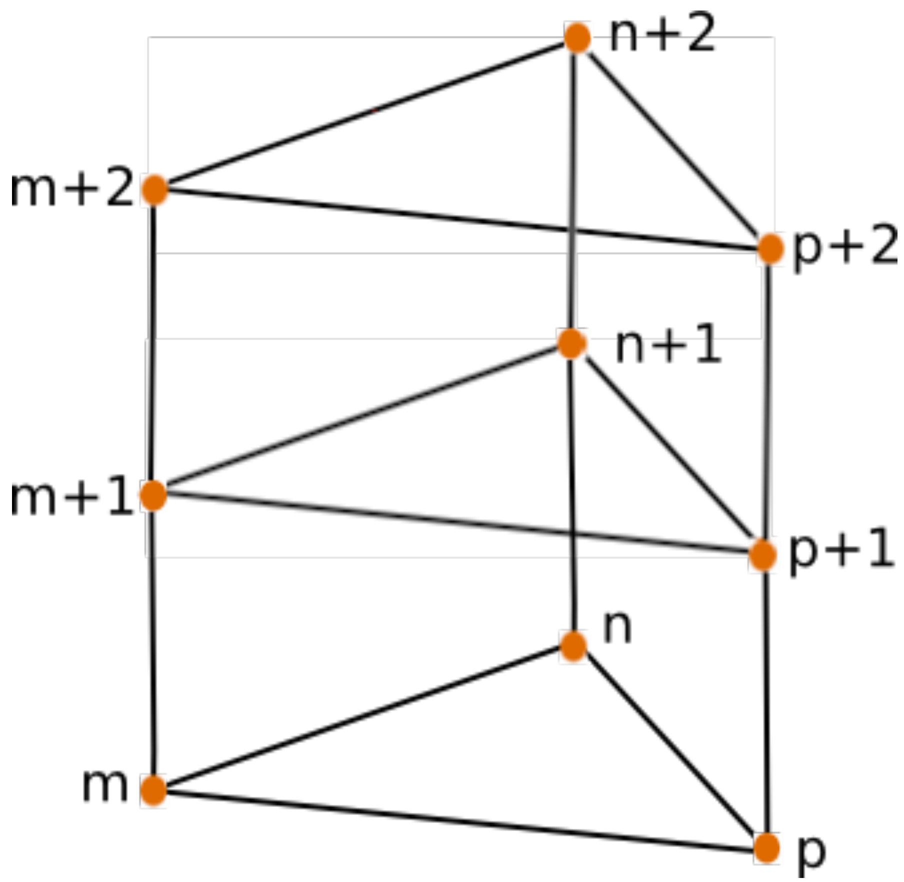
Also good for the performance side!

Vertical Alignment



Also good for the performance side!

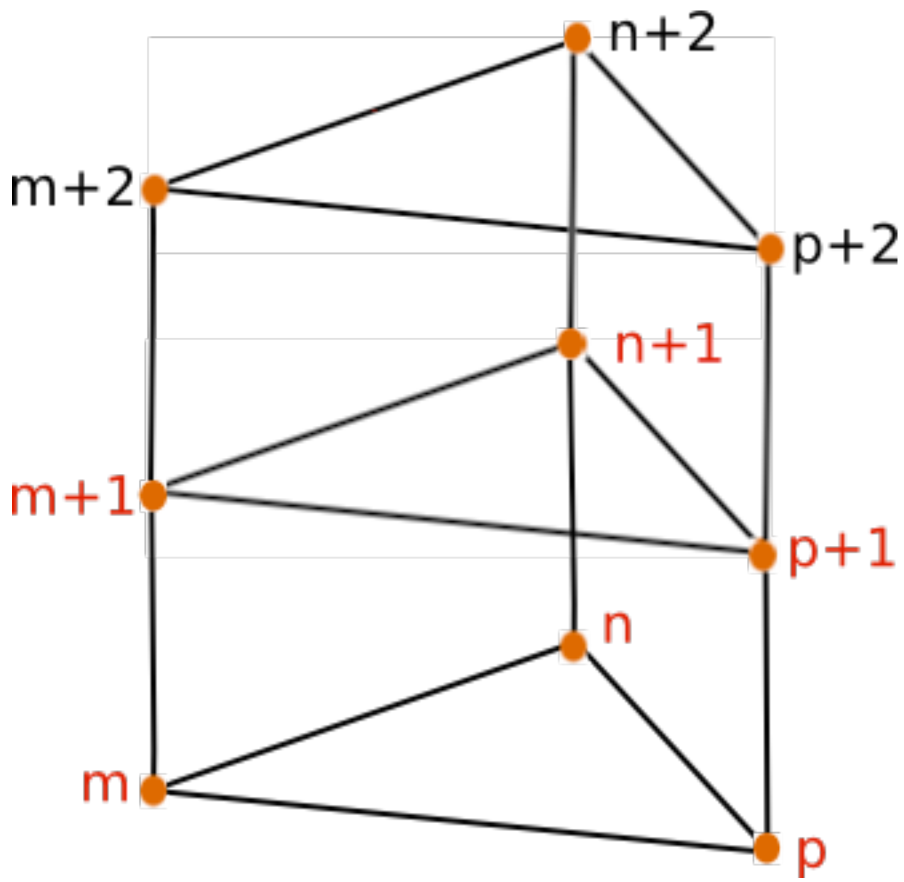
Vertical Alignment



<u>Also good for the performance side!</u>

Goal: Vertical alignment of cells does not require any explicit topological information to be maintained about the layers.

Vertical Alignment

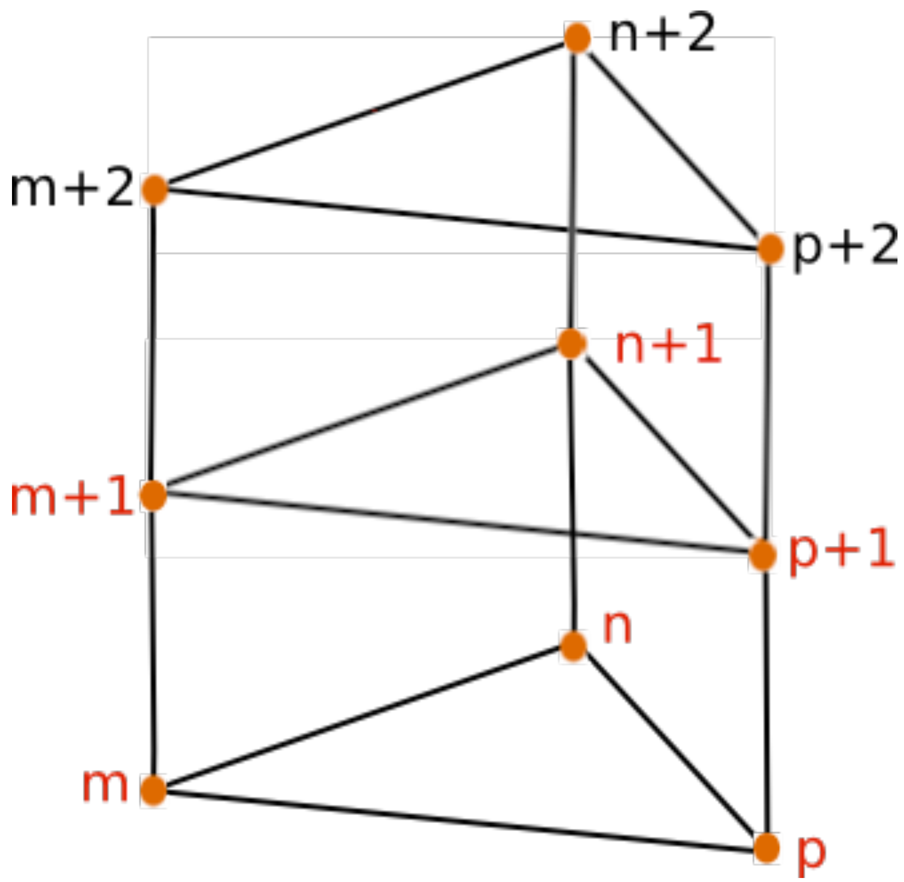


data[map[cell#]]

We can therefore:

➡ confine <u>indirect</u> accesses to the bottom layer of the mesh.

Vertical Alignment



data[[m, m+1,

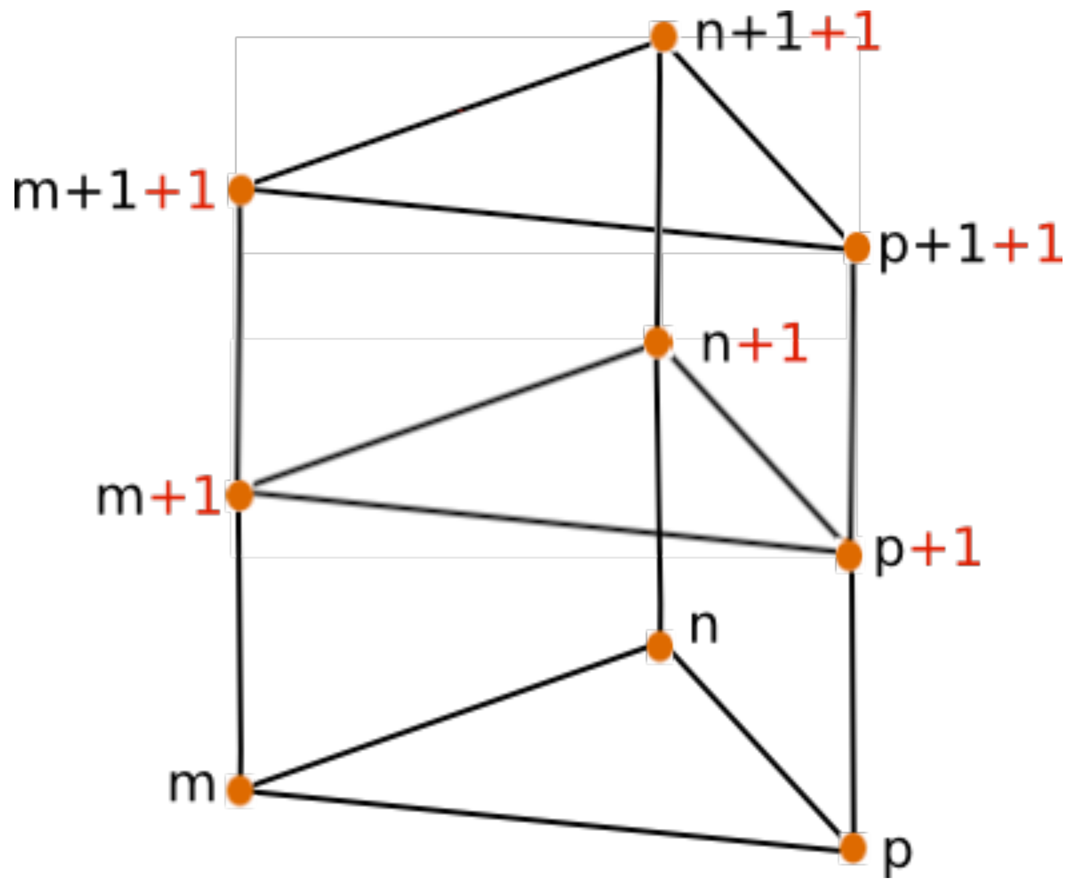
n, n+1,

p, p+1]]

We can therefore:

➡confine indirect accesses to the bottom layer of the mesh.

Vertical Alignment



data[[m+1, m+1+1,

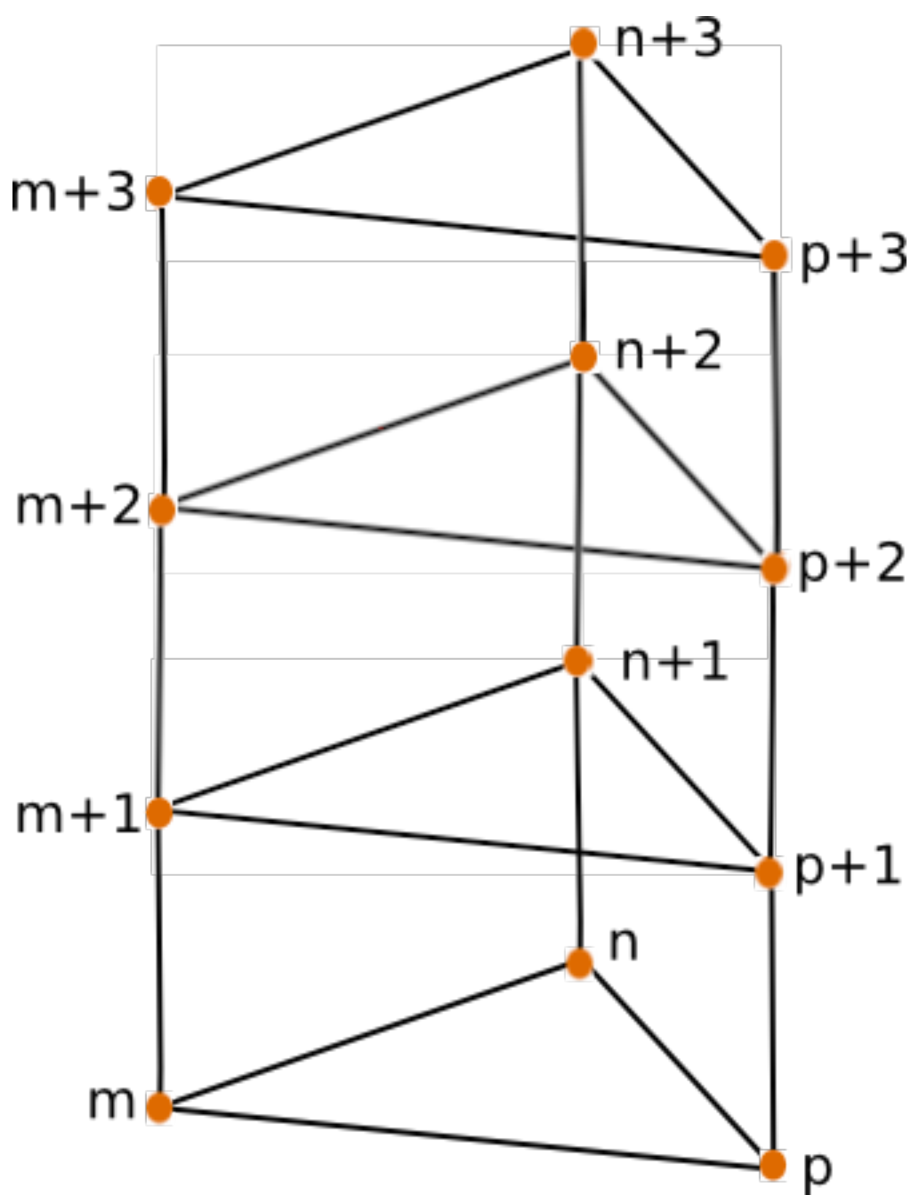
n+1, n+1+1,

p+1, p+1+1]]

We can therefore:

- ➡confine indirect accesses to the bottom layer of the mesh.
- ➡have the rest of the accesses as direct.

Vertical Alignment



data[[m+2, m+1+2,

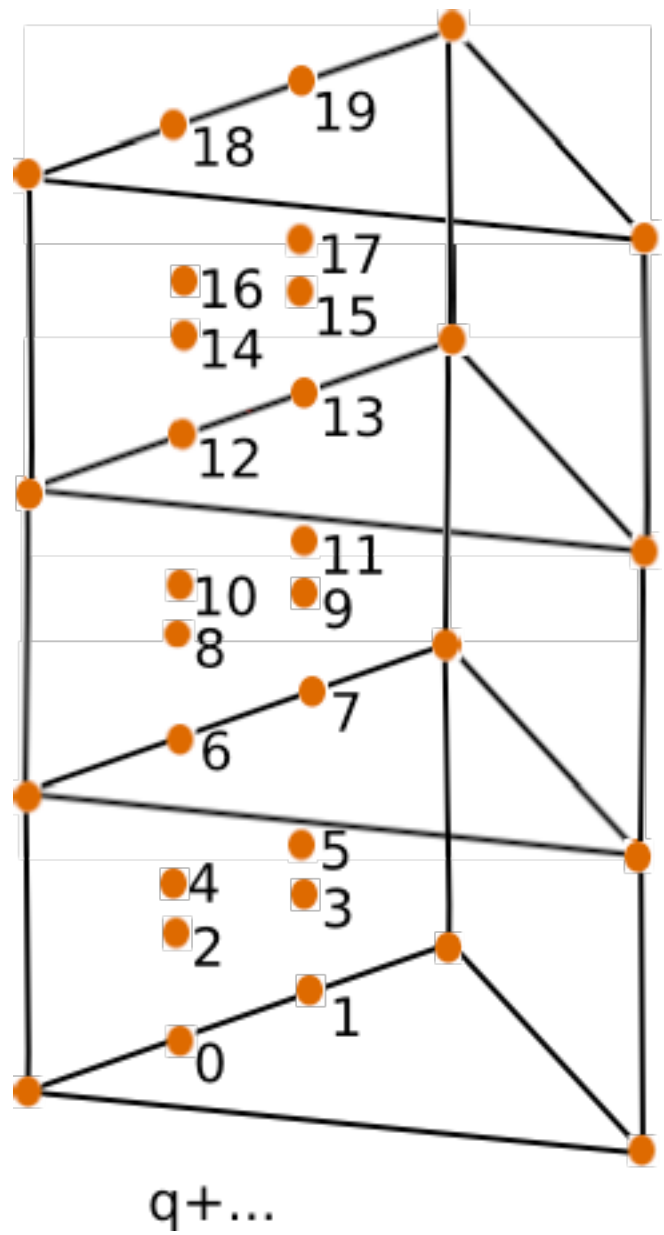
n+2, n+1+2,

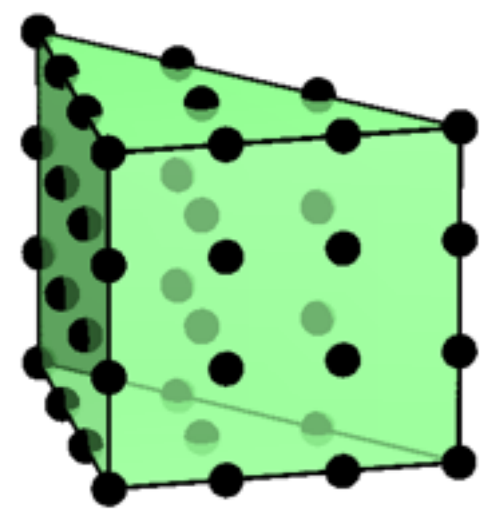
p+2, p+1+2]]

We can therefore:

- ➡confine <u>indirect</u> accesses to the bottom layer of the mesh.
- ➡have the rest of the accesses as <u>direct</u>.

Vertical Alignment





We can therefore:

- ➡confine indirect accesses to the bottom layer of the mesh.
- ➡have the rest of the accesses as direct.

Code generation

```c
void wrap_CG1xCG1__(int start, int end,
                    double* arg0_0, int *arg0_0_map0_0, int *arg0_0_off0_0 , int layer) {
    int xtr_arg0_0_map0_0[6];
    for (int n = start; n < end; n++ ) {
        int i = n;
        xtr_arg0_0_map0_0[0] = *(arg0_0_map0_0 + i * 6 + 0);
        xtr_arg0_0_map0_0[1] = *(arg0_0_map0_0 + i * 6 + 1);
        xtr_arg0_0_map0_0[2] = *(arg0_0_map0_0 + i * 6 + 2);
        xtr_arg0_0_map0_0[3] = *(arg0_0_map0_0 + i * 6 + 3);
        xtr_arg0_0_map0_0[4] = *(arg0_0_map0_0 + i * 6 + 4);
        xtr_arg0_0_map0_0[5] = *(arg0_0_map0_0 + i * 6 + 5);
        for (int j_0=0; j_0<layer; ++j_0){

            // Kernel call
            CG1xCG1(xtr_arg0_0_map0_0);

            xtr_arg0_0_map0_0[0] += 1;
            xtr_arg0_0_map0_0[1] += 1;
            xtr_arg0_0_map0_0[2] += 1;
            xtr_arg0_0_map0_0[3] += 1;
            xtr_arg0_0_map0_0[4] += 1;
            xtr_arg0_0_map0_0[5] += 1;
        }
    }
}
```

Code generation

```
void wrap_CG1xCG1__(int start, int end,
                    double* arg0_0, int *arg0_0_map0_0, int *arg0_0_off0_0 , int layer) {
    int xtr_arg0_0_map0_0[6];
    for (int n = start; n < end; n++ ) {
        int i = n;
        xtr_arg0_0_map0_0[0] = *(arg0_0_map0_0 + i * 6 + 0);
        xtr_arg0_0_map0_0[1] = *(arg0_0_map0_0 + i * 6 + 1);
        xtr_arg0_0_map0_0[2] = *(arg0_0_map0_0 + i * 6 + 2);
        xtr_arg0_0_map0_0[3] = *(arg0_0_map0_0 + i * 6 + 3);
        xtr_arg0_0_map0_0[4] = *(arg0_0_map0_0 + i * 6 + 4);
        xtr_arg0_0_map0_0[5] = *(arg0_0_map0_0 + i * 6 + 5);
        for (int j_0=0; j_0<layer; ++j_0){

            // Kernel call
            CG1xCG1(xtr_arg0_0_map0_0);

            xtr_arg0_0_map0_0[0] += 1;
            xtr_arg0_0_map0_0[1] += 1;
            xtr_arg0_0_map0_0[2] += 1;
            xtr_arg0_0_map0_0[3] += 1;
            xtr_arg0_0_map0_0[4] += 1;
            xtr_arg0_0_map0_0[5] += 1;
        }
    }
}
```

Loop over all cells

Code generation

```
void wrap_CG1xCG1__(int start, int end,
                    double* arg0_0, int *arg0_0_map0_0, int *arg0_0_off0_0 , int layer) {
    int xtr_arg0_0_map0_0[6];
    for (int n = start; n < end; n++ ) {
        int i = n;
        xtr_arg0_0_map0_0[0] = *(arg0_0_map0_0 + i * 6 + 0);
        xtr_arg0_0_map0_0[1] = *(arg0_0_map0_0 + i * 6 + 1);
        xtr_arg0_0_map0_0[2] = *(arg0_0_map0_0 + i * 6 + 2);
        xtr_arg0_0_map0_0[3] = *(arg0_0_map0_0 + i * 6 + 3);
        xtr_arg0_0_map0_0[4] = *(arg0_0_map0_0 + i * 6 + 4);
        xtr_arg0_0_map0_0[5] = *(arg0_0_map0_0 + i * 6 + 5);
        for (int j_0=0; j_0<layer; ++j_0){

            // Kernel call
            CG1xCG1(xtr_arg0_0_map0_0);

            xtr_arg0_0_map0_0[0] += 1;
            xtr_arg0_0_map0_0[1] += 1;
            xtr_arg0_0_map0_0[2] += 1;
            xtr_arg0_0_map0_0[3] += 1;
            xtr_arg0_0_map0_0[4] += 1;
            xtr_arg0_0_map0_0[5] += 1;
        }
    }
}
```

Loop over all cells

Use bottom map

Code generation

```
void wrap_CG1xCG1__(int start, int end,
                    double* arg0_0, int *arg0_0_map0_0, int *arg0_0_off0_0 , int layer) {
    int xtr_arg0_0_map0_0[6];
    for (int n = start; n < end; n++ ) {
        int i = n;
        xtr_arg0_0_map0_0[0] = *(arg0_0_map0_0 + i * 6 + 0);
        xtr_arg0_0_map0_0[1] = *(arg0_0_map0_0 + i * 6 + 1);
        xtr_arg0_0_map0_0[2] = *(arg0_0_map0_0 + i * 6 + 2);
        xtr_arg0_0_map0_0[3] = *(arg0_0_map0_0 + i * 6 + 3);
        xtr_arg0_0_map0_0[4] = *(arg0_0_map0_0 + i * 6 + 4);
        xtr_arg0_0_map0_0[5] = *(arg0_0_map0_0 + i * 6 + 5);
        for (int j_0=0; j_0<layer; ++j_0){

            // Kernel call
            CG1xCG1(xtr_arg0_0_map0_0);

            xtr_arg0_0_map0_0[0] += 1;
            xtr_arg0_0_map0_0[1] += 1;
            xtr_arg0_0_map0_0[2] += 1;
            xtr_arg0_0_map0_0[3] += 1;
            xtr_arg0_0_map0_0[4] += 1;
            xtr_arg0_0_map0_0[5] += 1;
        }
    }
}
```

Loop over all cells

Use bottom map

Loop over layers

Code generation

```c
void wrap_CG1xCG1__(int start, int end,
                    double* arg0_0, int *arg0_0_map0_0, int *arg0_0_off0_0 , int layer) {
    int xtr_arg0_0_map0_0[6];
    for (int n = start; n < end; n++ ) {
        int i = n;
        xtr_arg0_0_map0_0[0] = *(arg0_0_map0_0 + i * 6 + 0);
        xtr_arg0_0_map0_0[1] = *(arg0_0_map0_0 + i * 6 + 1);
        xtr_arg0_0_map0_0[2] = *(arg0_0_map0_0 + i * 6 + 2);
        xtr_arg0_0_map0_0[3] = *(arg0_0_map0_0 + i * 6 + 3);
        xtr_arg0_0_map0_0[4] = *(arg0_0_map0_0 + i * 6 + 4);
        xtr_arg0_0_map0_0[5] = *(arg0_0_map0_0 + i * 6 + 5);
        for (int j_0=0; j_0<layer; ++j_0){

            // Kernel call
            CG1xCG1(xtr_arg0_0_map0_0);

            xtr_arg0_0_map0_0[0] += 1;
            xtr_arg0_0_map0_0[1] += 1;
            xtr_arg0_0_map0_0[2] += 1;
            xtr_arg0_0_map0_0[3] += 1;
            xtr_arg0_0_map0_0[4] += 1;
            xtr_arg0_0_map0_0[5] += 1;
        }
    }
}
```

Loop over all cells

Use bottom map

Loop over layers

Add offset to get next cell up

Code generation

```c
void wrap_expression_1(int start, int end,
                       double * arg0_0, int *arg0_0_map0_0, double *arg1_0, int *arg1_0_map0_0,
                       int *_arg0_0_off0_0, int *_arg1_0_off0_0 , int layer) {
    double *arg1_0_vec[6];
    int xtr_arg0_0_map0_0[6];
    for ( int n = start; n < end; n++ ) {
        int i = n;
        arg1_0_vec[0] = arg1_0 + (arg1_0_map0_0[i * 6 + 0])* 3;
        arg1_0_vec[1] = arg1_0 + (arg1_0_map0_0[i * 6 + 1])* 3;
        arg1_0_vec[2] = arg1_0 + (arg1_0_map0_0[i * 6 + 2])* 3;
        arg1_0_vec[3] = arg1_0 + (arg1_0_map0_0[i * 6 + 3])* 3;
        arg1_0_vec[4] = arg1_0 + (arg1_0_map0_0[i * 6 + 4])* 3;
        arg1_0_vec[5] = arg1_0 + (arg1_0_map0_0[i * 6 + 5])* 3;
        xtr_arg0_0_map0_0[0] = *(arg0_0_map0_0 + i * 6 + 0);
        xtr_arg0_0_map0_0[1] = *(arg0_0_map0_0 + i * 6 + 1);
        xtr_arg0_0_map0_0[2] = *(arg0_0_map0_0 + i * 6 + 2);
        xtr_arg0_0_map0_0[3] = *(arg0_0_map0_0 + i * 6 + 3);
        xtr_arg0_0_map0_0[4] = *(arg0_0_map0_0 + i * 6 + 4);
        xtr_arg0_0_map0_0[5] = *(arg0_0_map0_0 + i * 6 + 5);
        for (int j_0=0; j_0<layer; ++j_0){
            double buffer_arg0_0[6] = {0};
            //Kernel call
            expression_kernel_1(buffer_arg0_0, arg1_0_vec);
            for (int i_0=0; i_0<6; ++i_0) {
                *(arg0_0 + (xtr_arg0_0_map0_0[i_0])*1) = buffer_arg0_0[i_0*1 + 0];
            }
            xtr_arg0_0_map0_0[0] += _arg0_0_off0_0[0];
            xtr_arg0_0_map0_0[1] += _arg0_0_off0_0[1];
            xtr_arg0_0_map0_0[2] += _arg0_0_off0_0[2];
            xtr_arg0_0_map0_0[3] += _arg0_0_off0_0[3];
            xtr_arg0_0_map0_0[4] += _arg0_0_off0_0[4];
            xtr_arg0_0_map0_0[5] += _arg0_0_off0_0[5];
            arg1_0_vec[0] += _arg1_0_off0_0[0] * 3;
            arg1_0_vec[1] += _arg1_0_off0_0[1] * 3;
            arg1_0_vec[2] += _arg1_0_off0_0[2] * 3;
            arg1_0_vec[3] += _arg1_0_off0_0[3] * 3;
            arg1_0_vec[4] += _arg1_0_off0_0[4] * 3;
            arg1_0_vec[5] += _arg1_0_off0_0[5] * 3;
        }
    }
}
```

Extruded Helmholtz Example

```
from firedrake import *
m = UnitSquareMesh(200, 200)
mesh = ExtrudedMesh(m, layers=layers, layer_height=1.0)
V = FunctionSpace(mesh, "CG", 2, vfamily="CG", vdegree=2)

# Define variational problem
u = TrialFunction(V)
v = TestFunction(V)

f = Function(V)
f.interpolate(Expression("(1+8*pi*pi)*cos(x[0]*pi*2)*cos(x[1]*pi*2)"))

a = (dot(grad(v), grad(u)) + v * u) * dx
L = f * v * dx

# This is RHS assembly
l = assemble(L)

# This is LHS assembly
A = assemble(a)
```

Extruded Helmholtz Example

```
from firedrake import *
m = UnitSquareMesh(200, 200)
mesh = ExtrudedMesh(m, layers=layers, layer_height=1.0)
V = FunctionSpace(mesh, "CG", 2, vfamily="CG", vdegree=2)

# Define variational problem
u = TrialFunction(V)
v = TestFunction(V)

f = Function(V)
f.interpolate(Expression("(1+8*pi*pi)*cos(x[0]*pi*2)*cos(x[1]*pi*2)"))

a = (dot(grad(v), grad(u)) + v * u) * dx
L = f * v * dx

# This is RHS assembly
l = assemble(L)

# This is LHS assembly
A = assemble(a)
```

Extruded Meshes in Firedrake

```
from firedrake import *

mesh = UnitSquareMesh(200, 200)

mesh = ExtrudedMesh(mesh, layers=100, layer_height=1.0, extrusion_type="uniform", kernel="...")
```

The type of extrusion

uniform: adds an extra coordinate for the new dimension the extrusion is going to be performed in. The vertical extent is evenly split between layers.

radial: usually applies to manifolds and extrudes the points of the mesh in the outwards direction from the origin.

Extruded Meshes in Firedrake

```
from firedrake import *

mesh = UnitSquareMesh(200, 200)

mesh = ExtrudedMesh(mesh, layers=100, layer_height=1.0, extrusion_type="uniform", kernel="...")
```

Custom extrusion

An extrusion kernel can be explicitly specified and passed to the extruded mesh constructor.

This overwrites both layer_height and extrusion_type.

Extruded Helmholtz Example

```
from firedrake import *
m = UnitSquareMesh(200, 200)
mesh = ExtrudedMesh(m, layers=layers, layer_height=1.0)
V = FunctionSpace(mesh, "CG", 2, vfamily="CG", vdegree=2)

# Define variational problem
u = TrialFunction(V)
v = TestFunction(V)

f = Function(V)
f.interpolate(Expression("(1+8*pi*pi)*cos(x[0]*pi*2)*cos(x[1]*pi*2)"))

a = (dot(grad(v), grad(u)) + v * u) * dx
L = f * v * dx

# This is RHS assembly
l = assemble(L)

# This is LHS assembly
A = assemble(a)
```

Valuable Bandwidth

$$\text{Valuable Bandwidth} = \frac{DV_{Total}}{\text{Execution time}}$$

We will use a simple metric to model the performance of bandwidth bound mesh wide operations i.e. the valuable bandwidth.

The goal: compare against the peak performance of the machine (STREAM).

Assumption: each piece of data required by the kernel is moved only once (assume perfect re-use).

Advantage: safeguard against bad numbering in the horizontal.

Limitation: in cases where data is shared the extra traffic generated by data being load more than once is not taken into account even though no better numbering exists.

Valuable Bandwidth

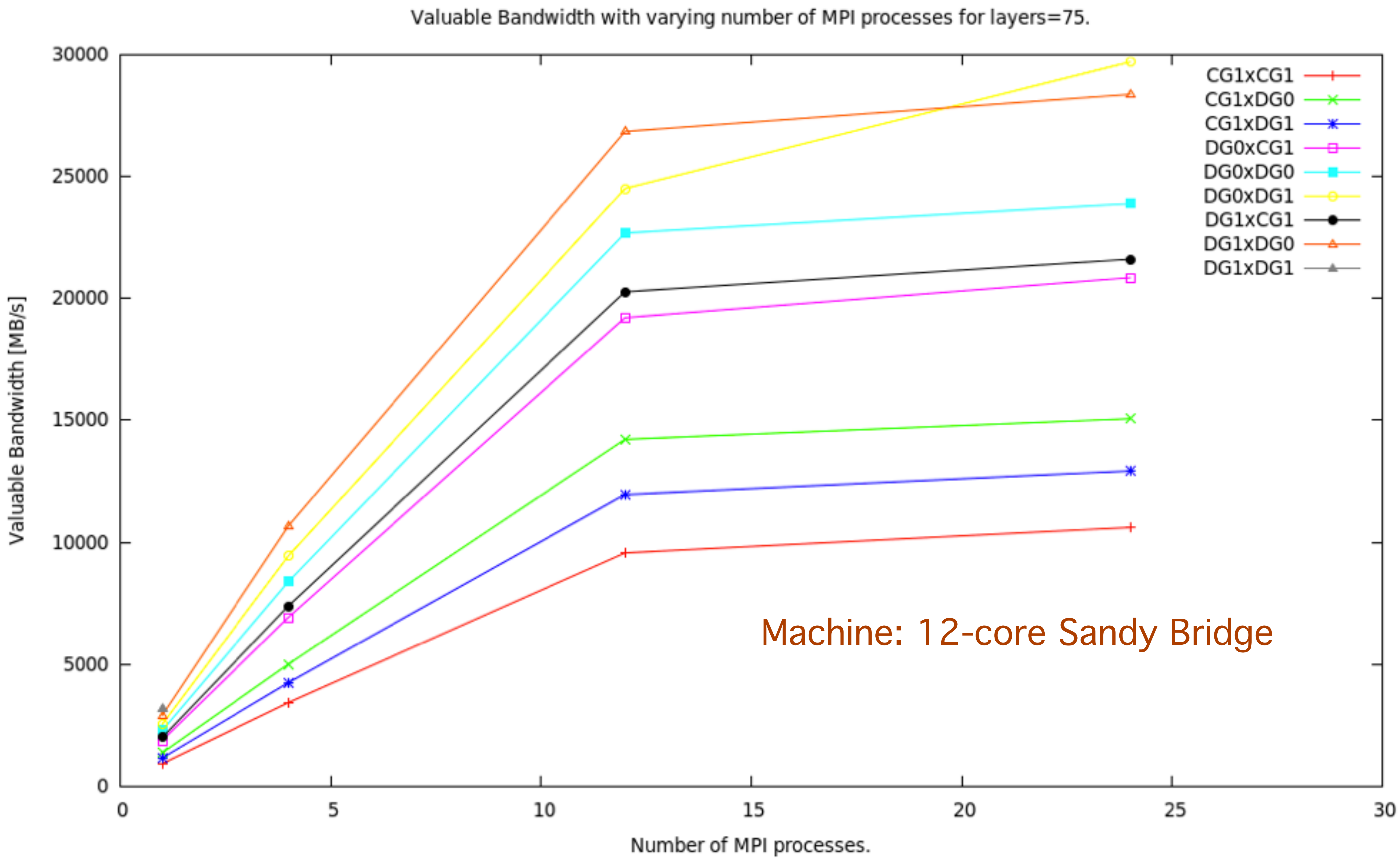
$$Valuable\ Bandwidth = \frac{DV_{Total}}{Execution\ time}$$

We will use a simple metric to model the performance of bandwidth bound mesh wide operations i.e. the valuable bandwidth.

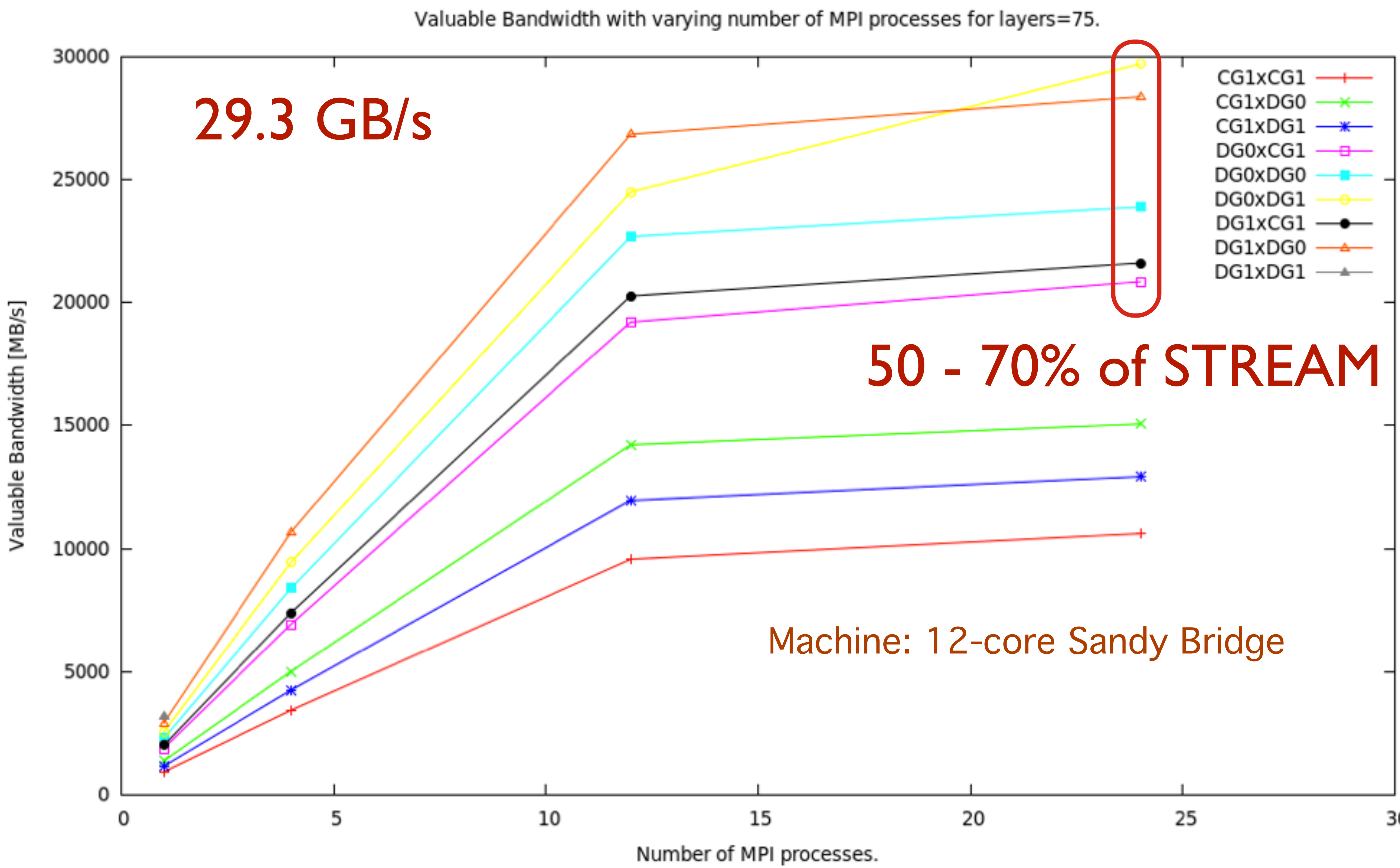
The goal: compare against the peak performance of the machine (STREAM).

Multithreaded STREAM performance: 42 GB/s

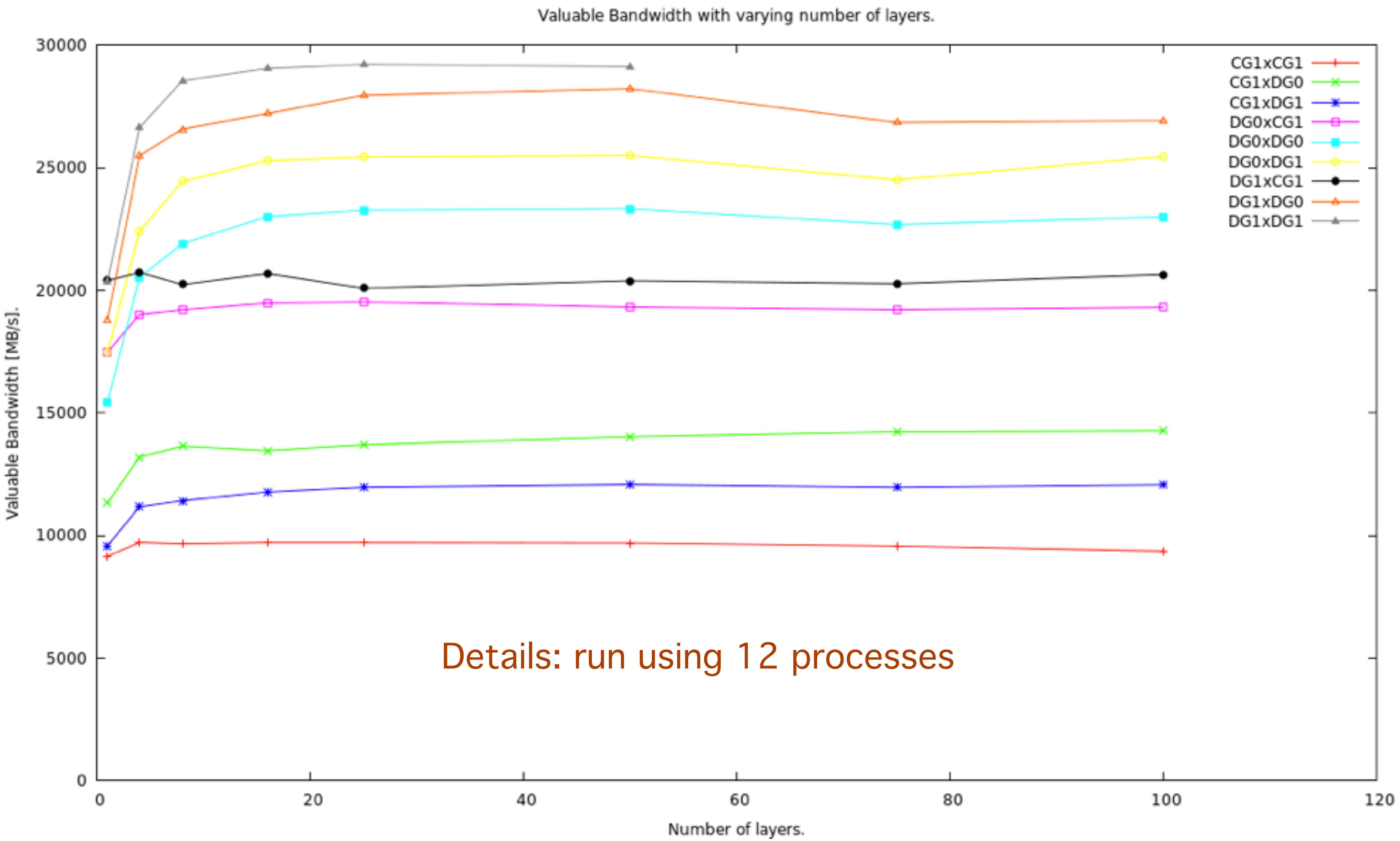
RHS Assembly Performance: v * dx



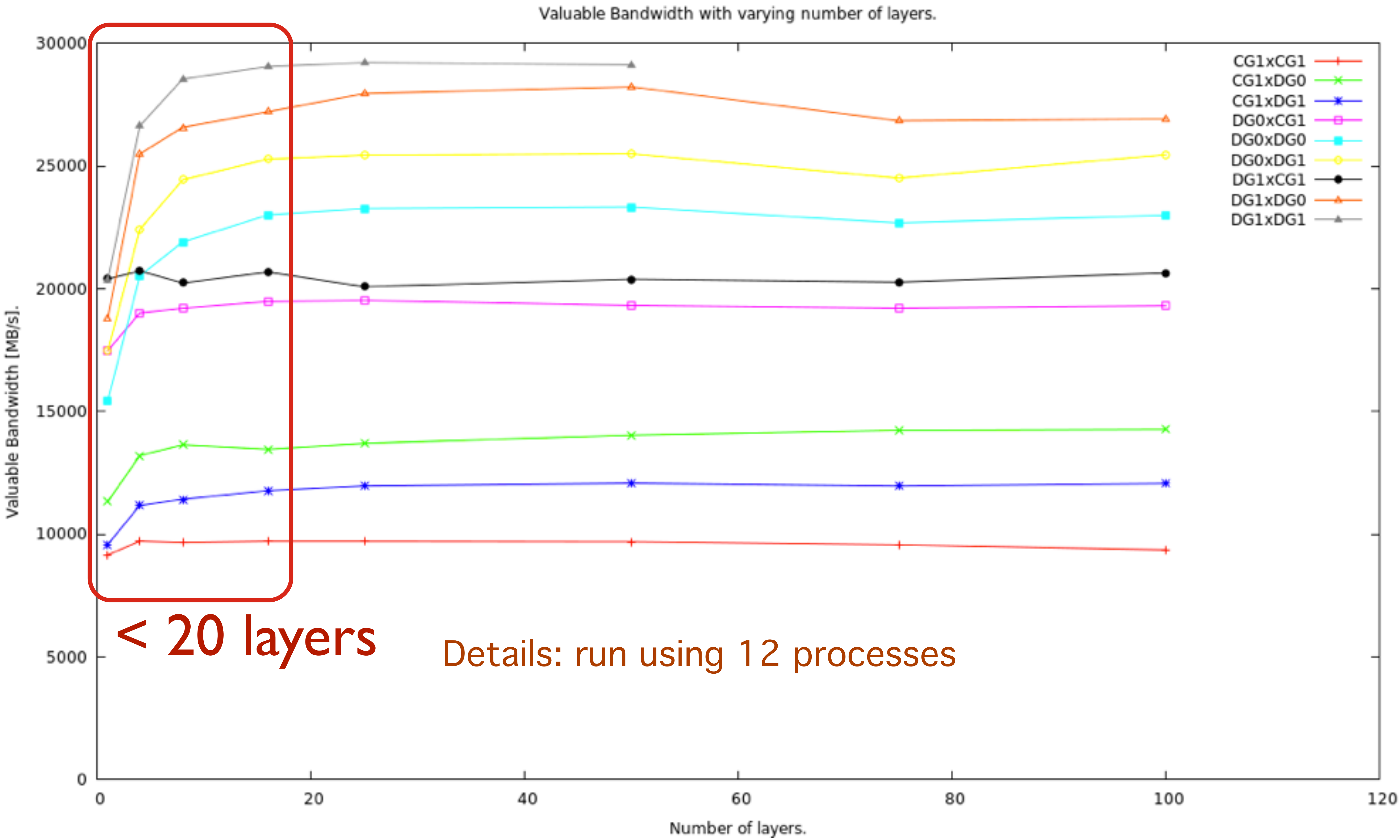
RHS Assembly Performance: v * dx



RHS Assembly Performance: v * dx

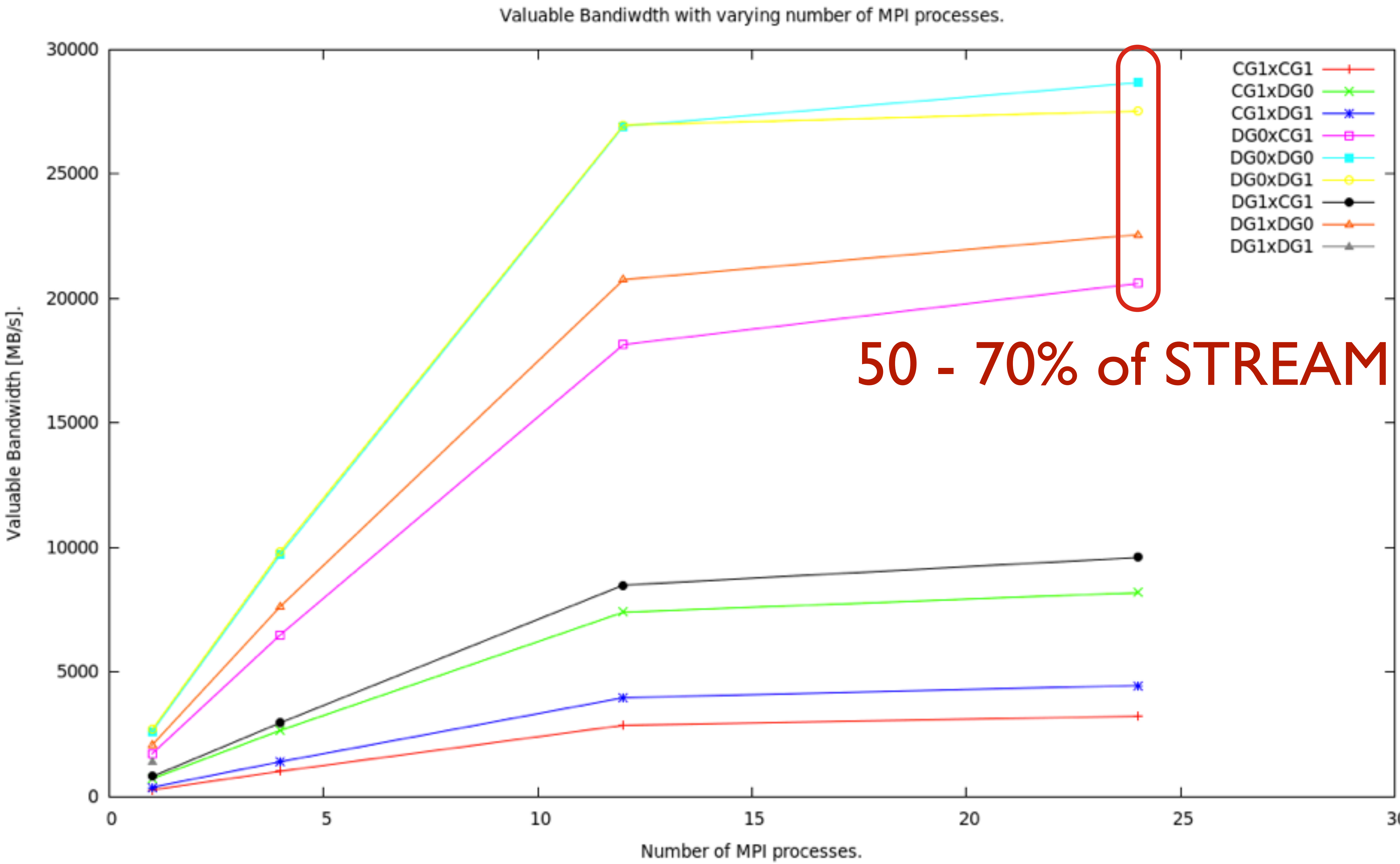


RHS Assembly Performance: v * dx

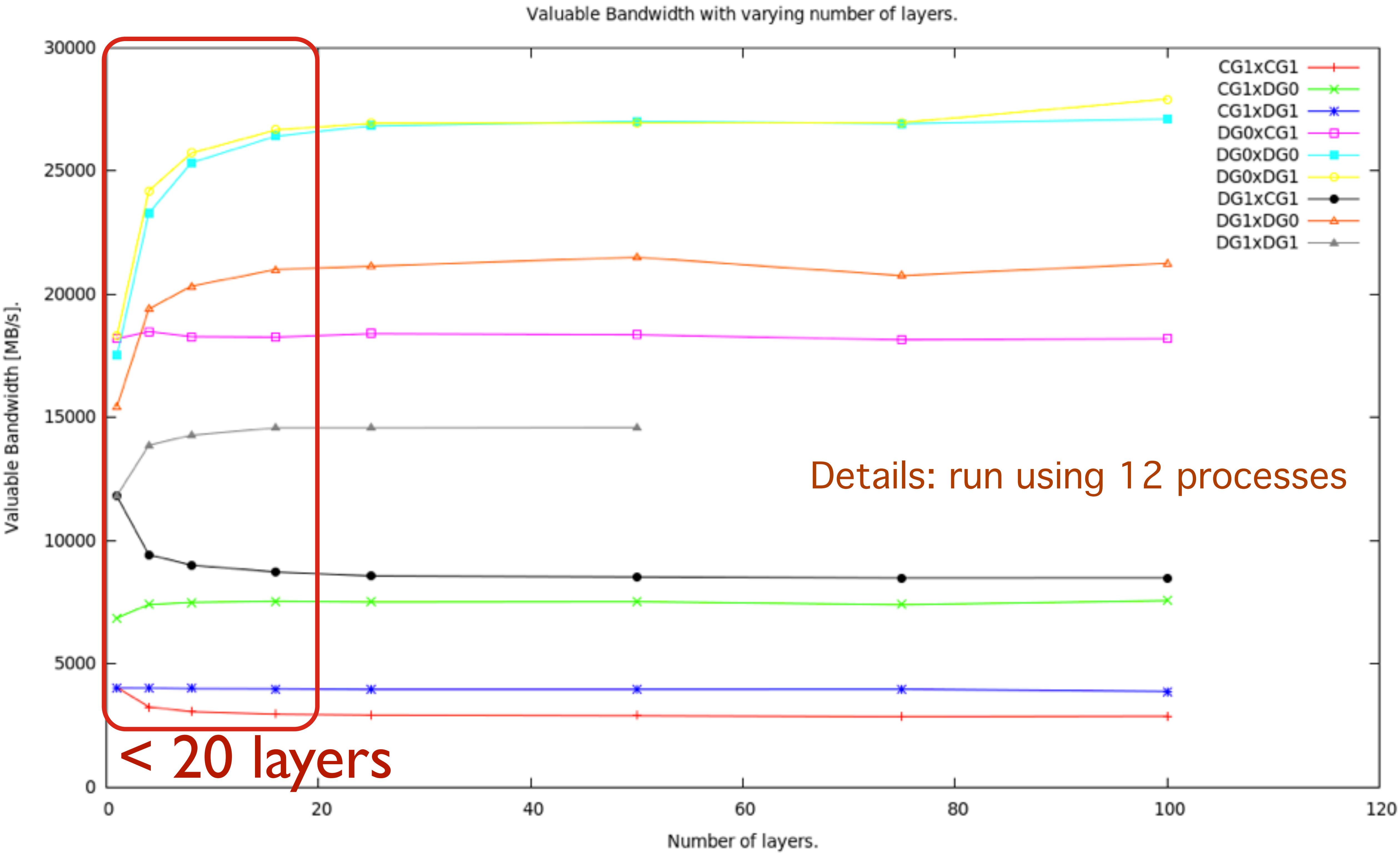


Details: run using 12 processes

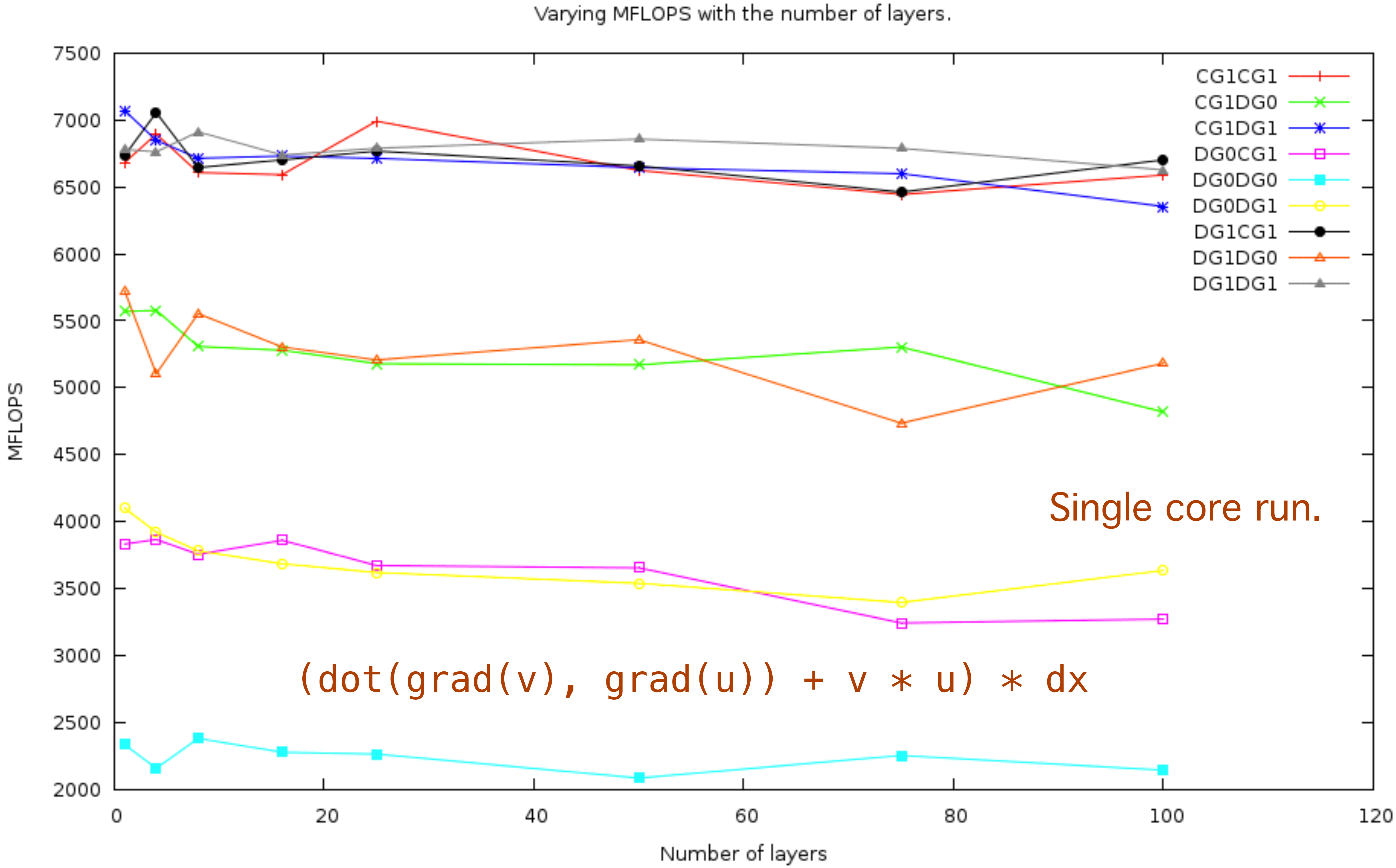
RHS Assembly Performance: f * v * dx



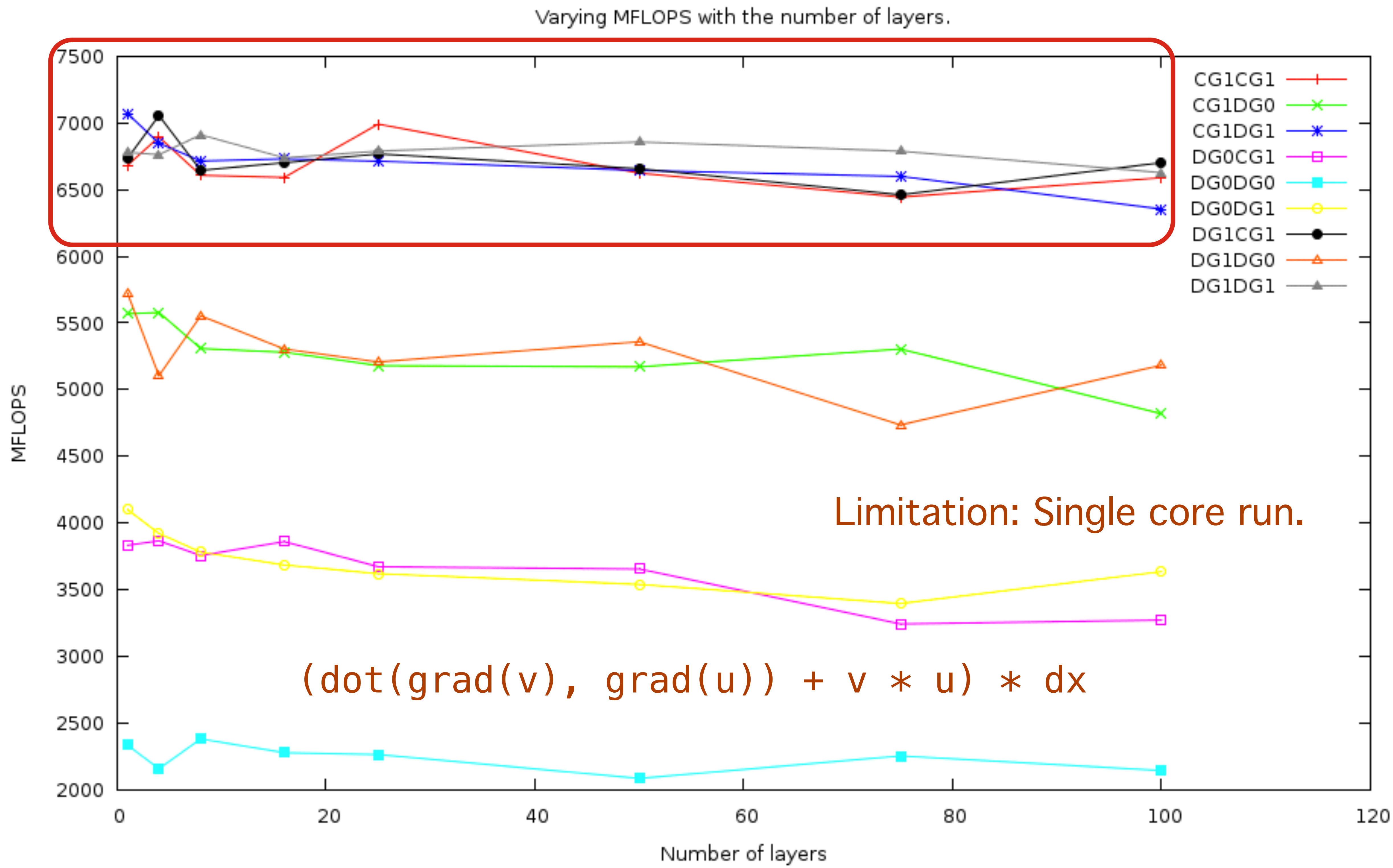
RHS Assembly Performance: f * v * dx



LHS Assembly Performance



LHS Assembly Performance



Future work: vertical structure exploitation in not just the kernel wrapper.

➡Improve the sparsity insertion to exploit the vertical structure: one insert per column.

➡Extend the valuable bandwidth model to include operational intensity: FLOPS per valuable bytes.


1948