STENFW – a stencil framework for compilers benchmarking

dmitry.mikushin@usi.ch

July 4, 2012

Description

STENFW is intended to ease the evaluation of different compiler suites like DSLs, automatic parallel code generators or conventional compilers. The framework is structured to keep the stencils implementations separated from the rest of the test suite, which handles MPI/CUDA management and visualization.

This is preview version. Comments on improving the code structure are highly appreciated!

Code structure

- **data2vdf** – data converter for visualizer
- **doc** – some documentation
- **generators** – inital data generators
- **grid** – the grid decomposition engine
- **stencils** – the stencils implementations, each stencil in its own subfolder
- **tests** – test applications, that use stencils
- **timing** – time measurement routines

Building

```
$ tar -xf stenfw_120620153943.tar.gz
$ cd stenfw
$ mkdir build
$ cd build/
$ cmake -DCMAKE_BUILD_TYPE="Release" ..
$ make
```

Build options

CMake build script supports the following mode switches:

- **HAVE_MPI** – enable or disable MPI support
- **HAVE_CUDA** – enable or disable CUDA support, performing explicit host-device data transfers by default, or add:
  - **HAVE_CUDA_MAPPED** – to use *host-mapped* memory directly accessible from the GPU kernels
  - **HAVE_CUDA_PINNED** – to perform explicit host-device data transfers in *pinned* memory
- **HAVE_VISUALIZE** – turn on the debug visualization based on UCAR Vapor, external runtime libraries needed

Example:

```
$ cmake -DHAVE_MPI=ON -DHAVE_CUDA=ON -DHAVE_CUDA_MAPPED=ON -DHAVE_VISUALIZE=ON↩
      -DCMAKE_BUILD_TYPE="Release" ..
$ make
```

Note the combination of **HAVE_MPI** and **HAVE_CUDA** currently can work only together with **HAVE_CUDA_MAPPED**

Test command line

The command line options change, depending on enabled features:

- Without MPI and GPU:

```
$ ./wave13pt
Usage: ./wave13pt <n> <nt>
where <n> and <nt> must be positive
```

- Without MPI, with GPU:

```
$ ./wave13pt
Usage: ./wave13pt <n> <nt> <CPU|GPU>
where <n> and <nt> must be positive
```

Test command line

The command line options change, depending on enabled features:

- With MPI, without GPU:

```
$ ./wave13pt
Usage: ./wave13pt <n> <nt> <sx> <sy> <ss>
where <n> and <nt> must be positive
and <sx> * <sy> * <ss> must equal to the number of MPI processes
```

- With MPI and GPU:

```
$ ./wave13pt
Usage: ./wave13pt <n> <nt> <sx> <sy> <ss> <CPU|GPU>
where <n> and <nt> must be positive
and <sx> * <sy> * <ss> must equal to the number of MPI processes
```

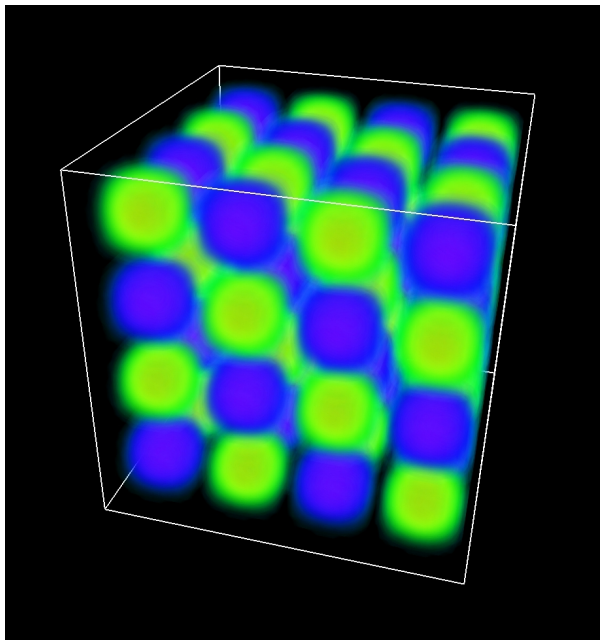
Testing example

Test *wave13pt* stencil on 2 GPUs used by 2 MPI processes:

```
$ cd bin/
$ mpirun -np 2 ./wave13pt 64 64 2 1 1 GPU
Mapping problem grid 64 x 64 x 64 onto 2 x 1 x 1 compute grid
step 0
...
min and max values of data output: -0.421547, 0.421548
```

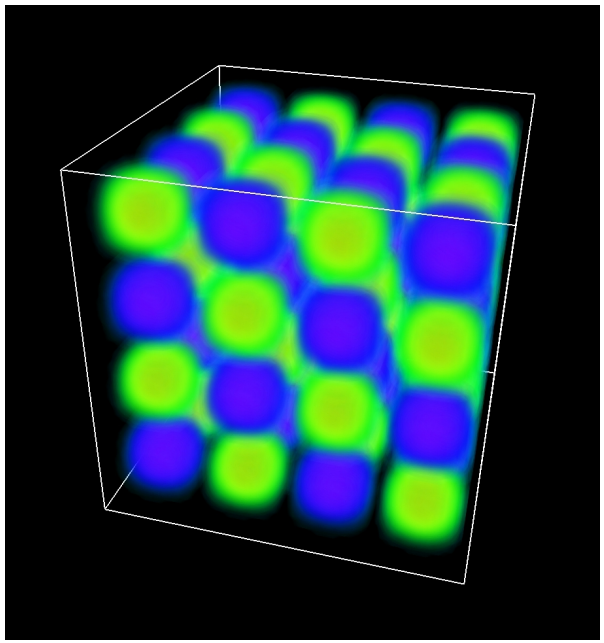
Visualizing with UCAR Vapor

1. Login with $ **ssh -X** to enable graphics forwarding
2. Execute /export/opt/vapor/vaporgui
3. Load the build/<stencil>.vdf dataset into the current session of UCAR Vapor.



Visualizing with UCAR Vapor

4. Select “3DTexture” for render type
5. Check “Instance1” box
6. Delete the default linear opacity widget and replace it with inverted Gaussian



Batch testing

STENFW comes with a tiny batch testing Perl script to ensure new changes do not break all supported configurations:

```
$ ./test
```

Debugging for correctness

- STENFW has a special stencil *isum13pt*, which tests the exact correctness of the parallel version result against serial version result on integer data
- Useful for debugging MPI or CUDA bugs
- Has embedded 2d visualization system to present the results difference (images will be placed into bin/ folder)

```
$ mpirun -np 2 ./isum13pt 64 64 1 1 2 CPU
Mapping problem grid 64 x 64 x 64 onto 1 x 1 x 2 compute grid
step 2 time = 0.005935042 sec
0.000000% different, max = 0 @ 262143
step 3 time = 0.005831204 sec
5.493164% different, max = 42 @ 134674
step 4 time = 0.005971761 sec
...
```

Testing on GraphIT!

```
$ tar -xf stenfw_120704165836.tar.gz
$ cd stenfw
$ mkdir build
$ cd build/
$ cmake -DHAVE_MPI=ON -DHAVE_CUDA=ON -DHAVE_CUDA_MAPPED=ON -DHAVE_VISUALIZE=ON -↩
    DCMAKE_BUILD_TYPE="Release" ..
$ OMPI_CC=gcc OMPI_CXX=g++ make
$ cleo-submit -np 2 ./test_wave13pt 64 64 2 1 1 CPU
$ cleo-submit -np 2 ./test_wave13pt 128 128 2 1 1 GPU
```

Exercise

Currently kernel in **wave13pt_gpu.cu** in invoked on a very simple compute grid, without blocks:

```
extern "C" int wave13pt_gpu(int nx, int ny, int ns,
const real c0, const real c1, const real c2,
real* w0, real* w1, real* w2)
{
wave13pt_gpu_kernel<<<dim3(nx - 4, ny - 4, 1), ns - 4>>>(nx, ny, ns, c0, c1, ↩
     c2, w0, w1, w2);
CUDA_SAFE_CALL(cudaDeviceSynchronize());
return 0;
}
```

Task for entering certification process: Reimplement kernel code and kernel invocation in a way that it could utilize compute grid blocks of *arbitary* size (parameter or preprocessor symbol BLOCK_SIZE), and the program in the same time could correctly handle *arbitary* problem dimensions. Example:

```
; BLOCK_SIZE=32
$ ./test_wave13pt 113 64 4 1 1 GPU
```