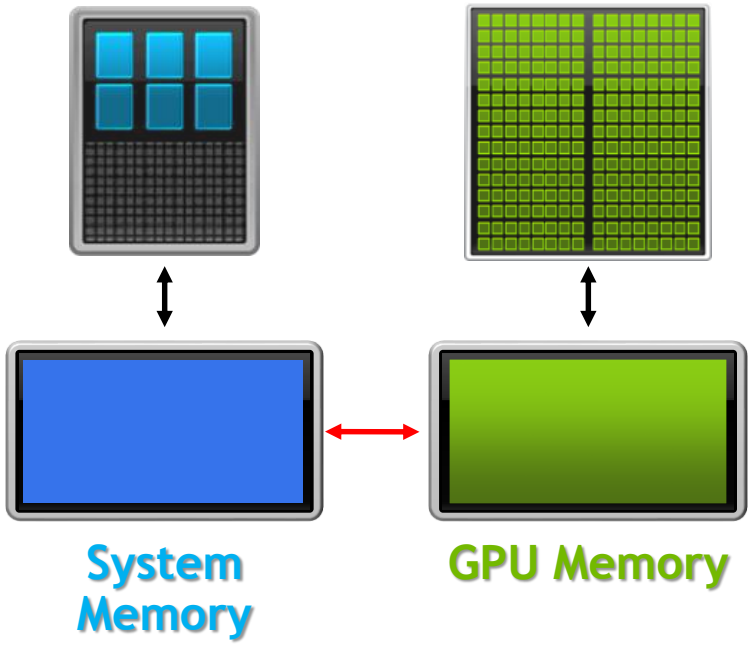
UNIFIED MEMORY ON KEPLER AND MAXWELL

Jiri Kraus, April 25th 2016

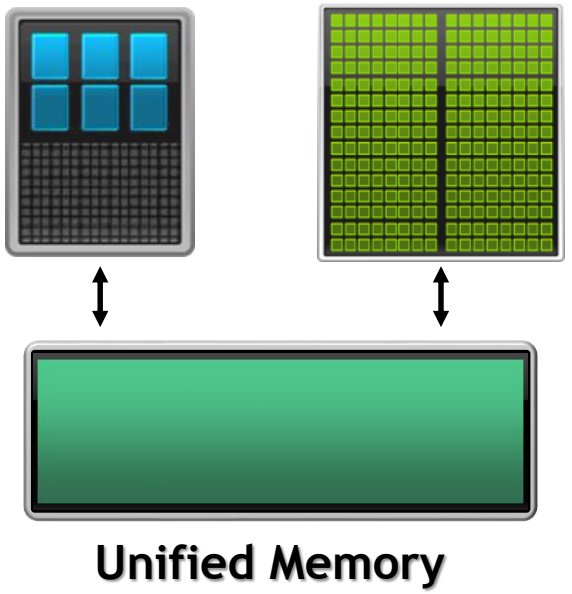
UNIFIED MEMORY

Dramatically Lower Developer Effort

Traditional Developer View



Developer View With Unified Memory



UNIFIED MEMORY

Traditional Developer View

```
void sortfile(FILE *fp, int N) {
  char *data;
  char *data_d;
  data = (char *)malloc(N);
  cudaMalloc( &data_d, N );

  fread(data, 1, N, fp);

  cudaMemcpy( data_d, data, N,
              cudaMemcpyHostToDevice);
  qsort<<<...>>>(data, N, 1, compare);

  cudaMemcpy( data, data_d, N,
              cudaMemcpyDeviceToHost);
  use_data(data);
  cudaFree(data_d); free(data); }
```

Developer View With Unified Memory

```
void sortfile(FILE *fp, int N) {
  char *data;

  cudaMallocManaged( &data, N );


  fread(data, 1, N, fp);



  qsort<<<...>>>(data, N, 1, compare);
  cudaDeviceSynchronize();


  use_data(data);
  cudaFree(data);                  }
```

UNIFIED MEMORY

Advantages

Unified Memory can be used in CPU and GPU code

No need for explicit device allocation(cudaMalloc) or memory copies (cudaMemcpy)

No need to fully understand data flow and allocation logic of application

Incremental profiler driven acceleration -> Data movement is just another optimization

UNIFIED MEMORY ON KEPLER AND MAXWELL

Implementations Details

Only heap and global data can be Unified Memory, e.g. no stack data:

```
char data[10];
qsort<<<...>>>(data                    pare);
```

Accessing Stack data from kernel not possible!

Data is coherent only at kernel launch and sync points

Its not allowed to access unified memory in host code while a kernel is running. Doing so may result in a segmentation fault.

CONCURRENT HOST DEVICE ACCESSES

CUDA_LAUNCH_BLOCKING

Its not allowed to access unified memory in host code while a kernel is running. Doing so may result in a segmentation fault:

```
ubuntu@ip-10-152-53-181:~/Pipelining$ ./pipelining
Bus error (core dumped)
```

CUDA_LAUNCH_BLOCKING can be used to check if missing synchronization is the issue:

```
ubuntu@ip-10-152-53-181:~/Pipelining$ CUDA_LAUNCH_BLOCKING=1 ./pipelining
Number of Primes = 151516
Runtime = 17.2459 ms
```

http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#env-vars

CONCURRENT HOST DEVICE ACCESSES

cuda-gdb

```
ubuntu@ip-10-152-53-181:~/Pipelining$ cuda-gdb ./pipelining
NVIDIA (R) CUDA Debugger
...
Reading symbols from /home/ubuntu/Pipelining/pipelining...(no debugging symbols
found)...done.
(cuda-gdb) run
Starting program: /home/ubuntu/Pipelining/./pipelining
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
[New Thread 0x7ffff5a0c700 (LWP 1451)]
[New Thread 0x7fffefffff700 (LWP 1452)]
[New Thread 0x7fffef7fe700 (LWP 1453)]

Program received signal CUDA_EXCEPTION_15, Invalid Managed Memory Access.
0x0000000000402b4f in generate_data(int*, int, int) ()
(cuda-gdb)
```

UNIFIED MEMORY ON KEPLER AND MAXWELL

Implementations Details

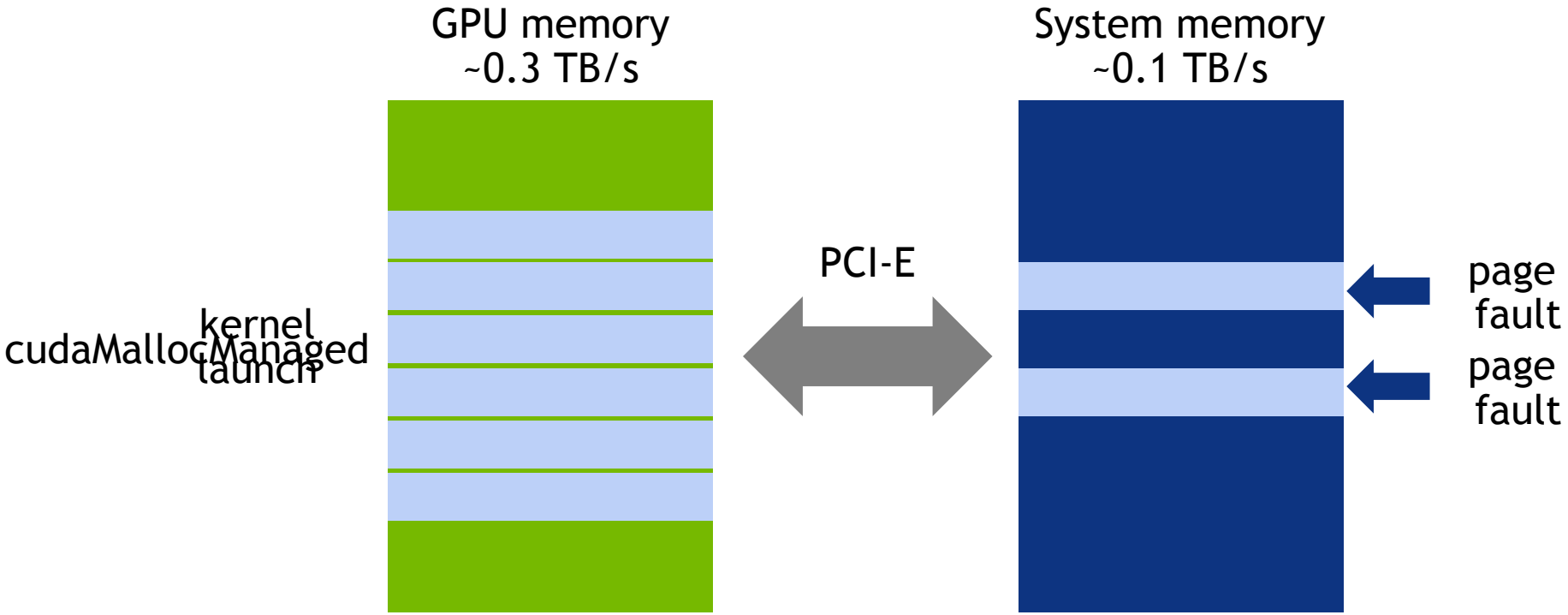
Allocation granularity of one page (4k) leads to allocation overhead (time and space) for small objects

Transfer overhead for small objects as data is transferred with the granularity of a single page

Number of allocations is limited to vm.max_map_count (defaults to 64k)

UNIFIED MEMORY ON KEPLER AND MAXWELL

Implementations Details



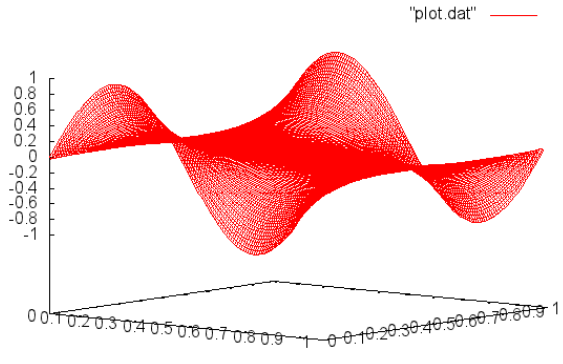
WEIGHTED JACOBI

Solves the 2D-Laplace equation on a rectangle

$$\Delta \boldsymbol{u}(\boldsymbol{x}, \boldsymbol{y}) = \mathbf{0} \; \forall \; (\boldsymbol{x}, \boldsymbol{y}) \in \Omega \backslash \boldsymbol{\delta}\Omega$$

Dirichlet boundary conditions (constant values on boundaries) on left and right boundary

Periodic boundary conditions on top and bottom boundary



WEIGHTED JACOBI

While not converged

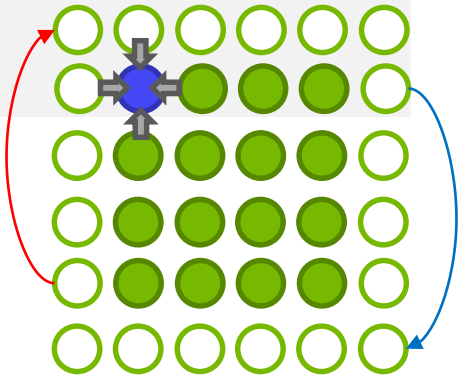
Do Jacobi step:

```
for (int iy=1; iy < ny-1; ++iy)

for (int ix=1; ix < nx-1; ++ix)

    float a_new_val = 0.25f*(a[ix-1][iy] + a[ix+1][iy] + a[ix][iy-1] + a[ix][iy+1]);

    a_new[ix][iy] = weight*a_new_val+(1.0f-weight)*a[ix][iy];
```

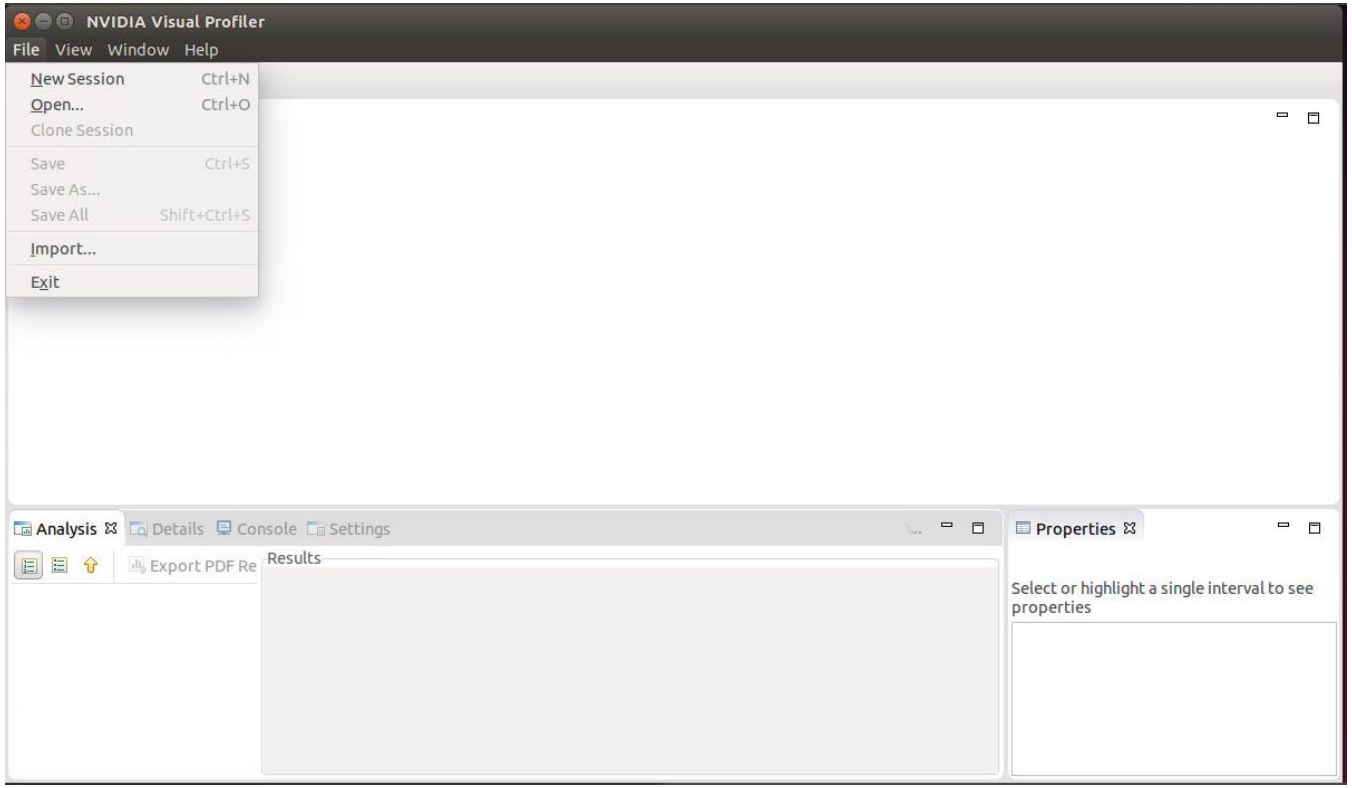
Swap `a_new` and `a`

Apply periodic boundary conditions to `a`

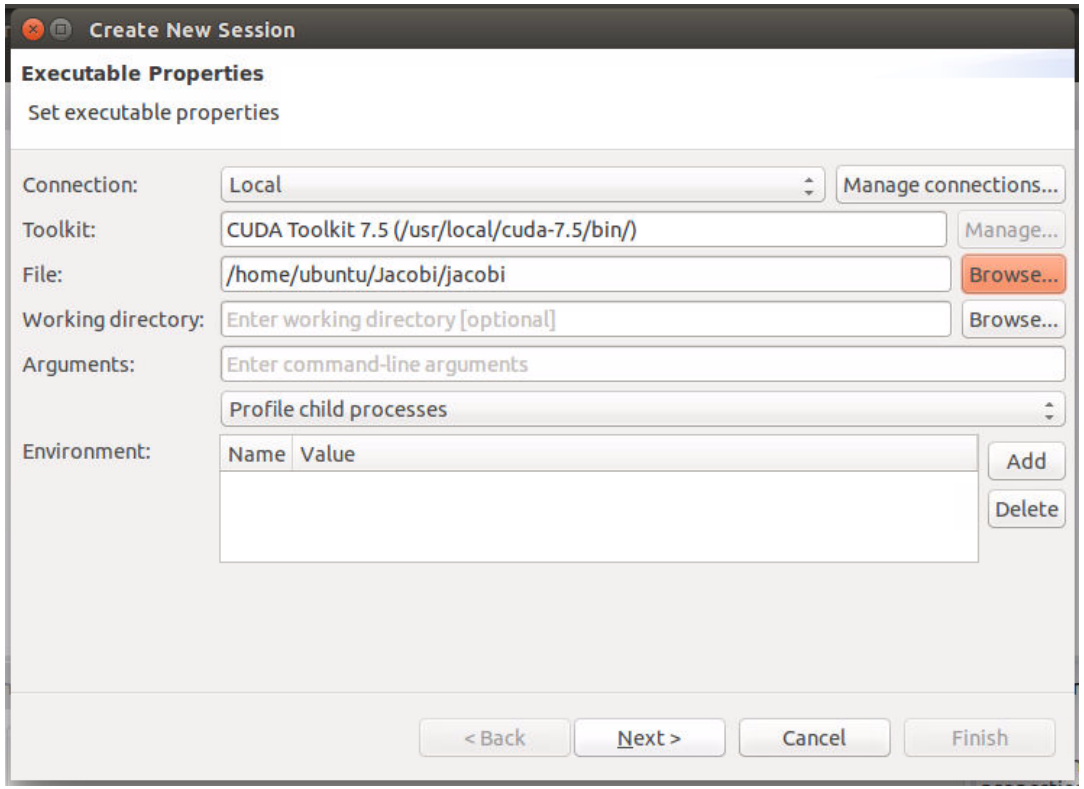
Next iteration



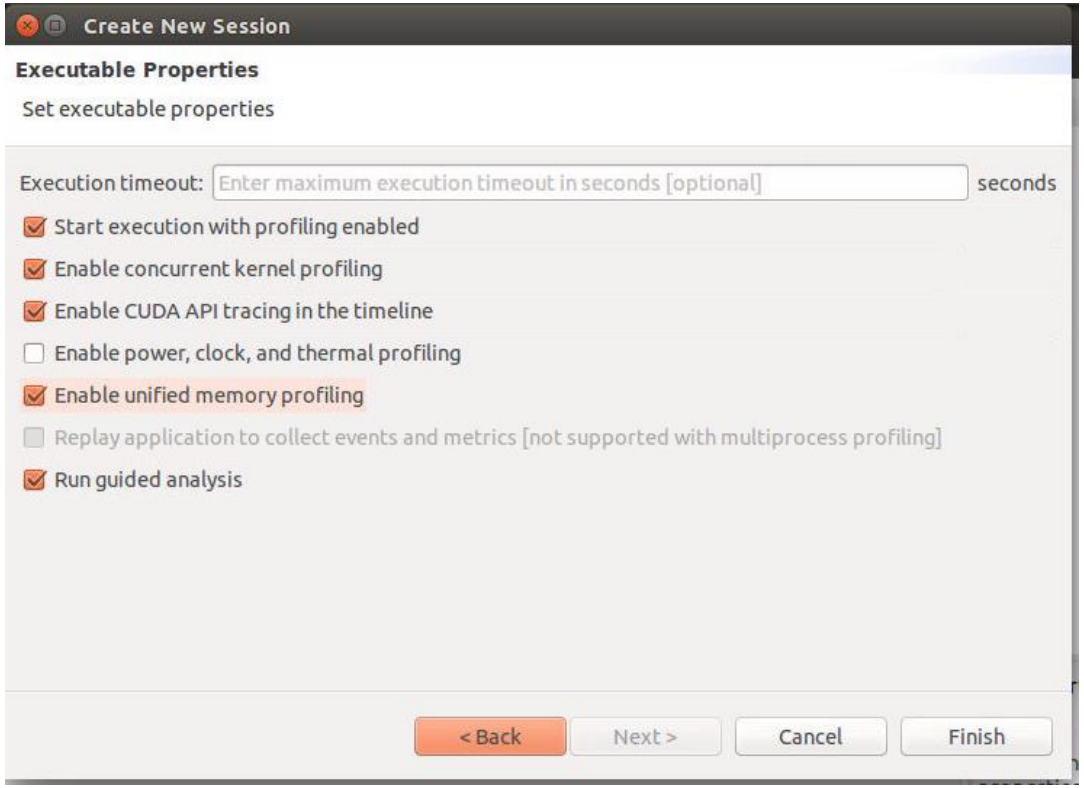
UNIFIED MEMORY PROFILING WITH NVVP



UNIFIED MEMORY PROFILING WITH NVVP

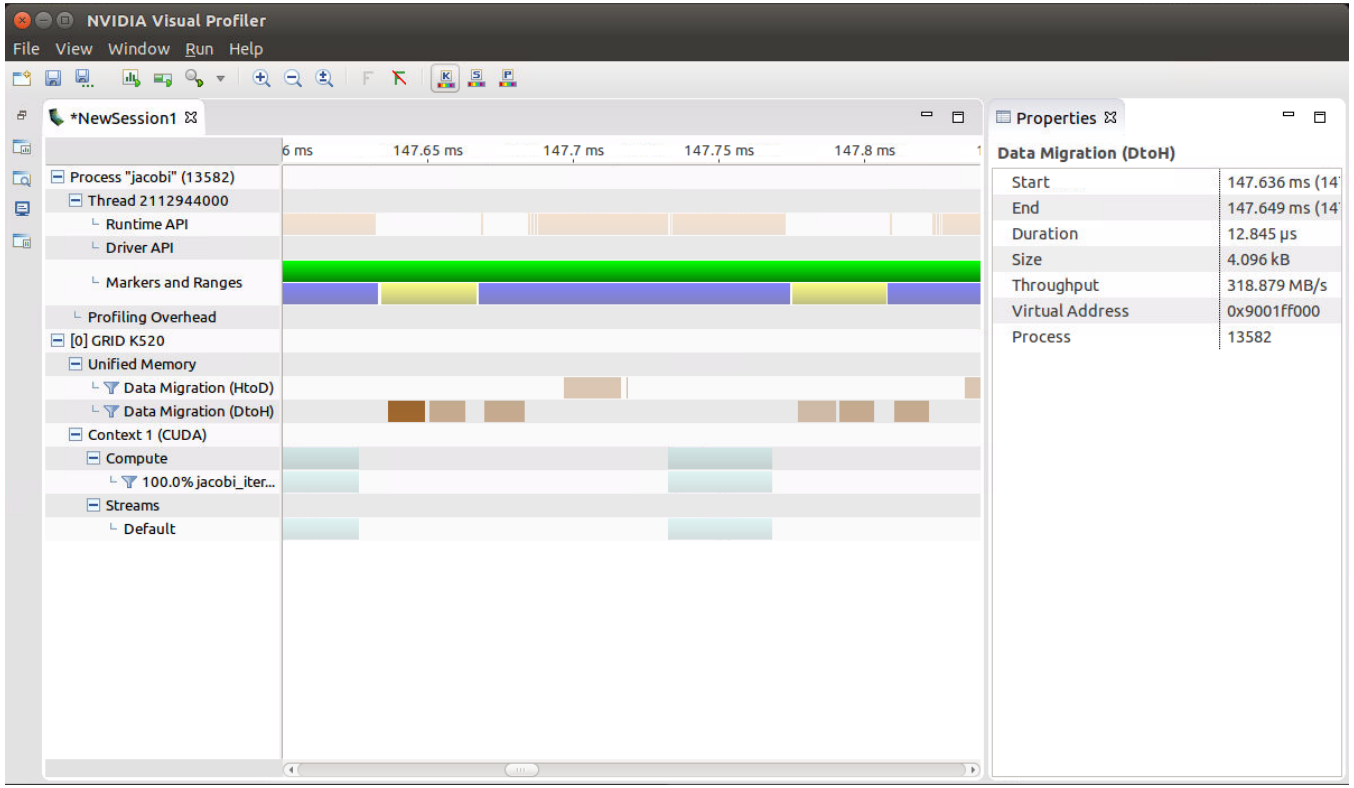


UNIFIED MEMORY PROFILING WITH NVVP



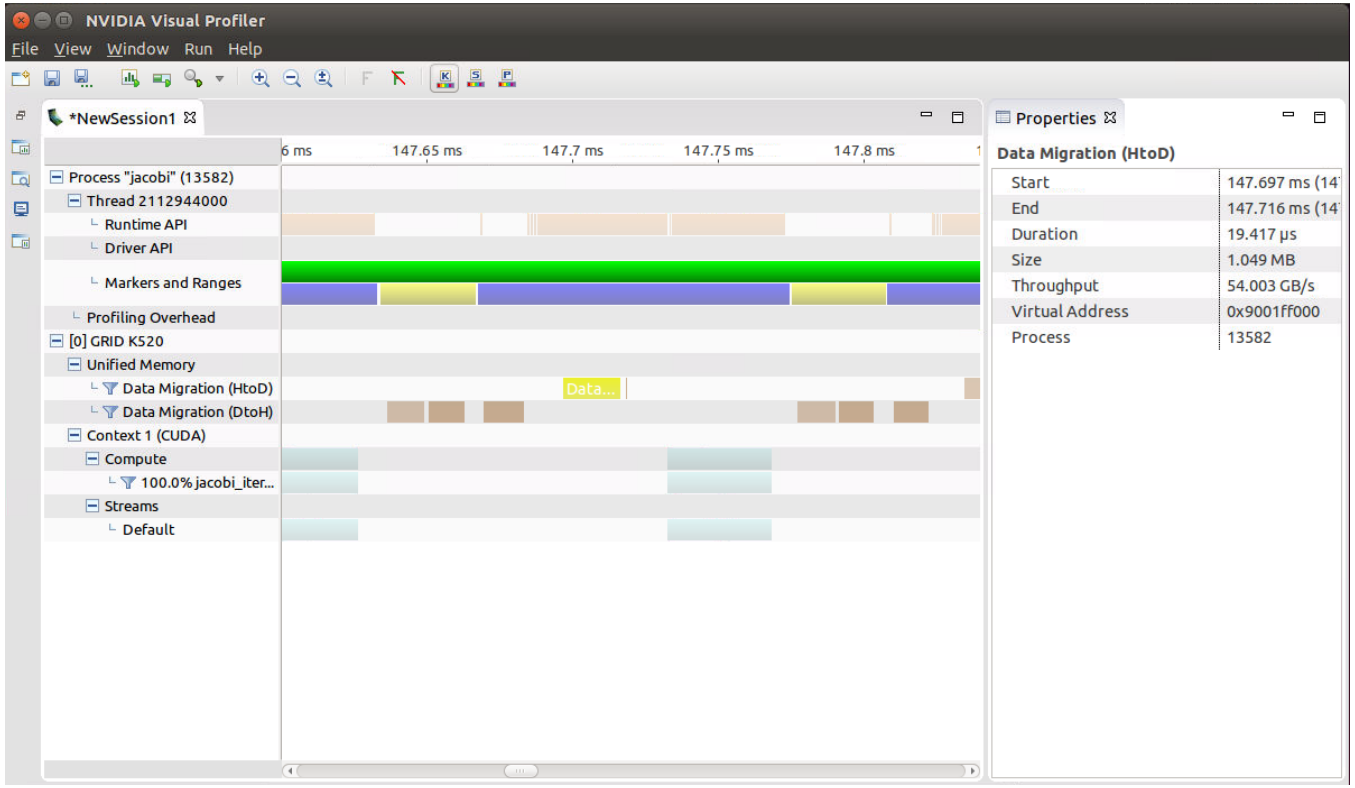
UNIFIED MEMORY PROFILING WITH NVVP

Host to Device (DtoH) Data Migration



UNIFIED MEMORY PROFILING WITH NVVP

Host to Device (HtoD) Data Migration



HANDS-ON

Task: Jacobi

Task: Avoid all data migrations within the while loop of the Jacobi solver

Apply boundary conditions with the provided GPU kernel (`apply_periodic_bc`)

Try to avoid the remaining data migrations? What's causing this?

Look for TODOs

Optional Task: What is the possible additional improvement?

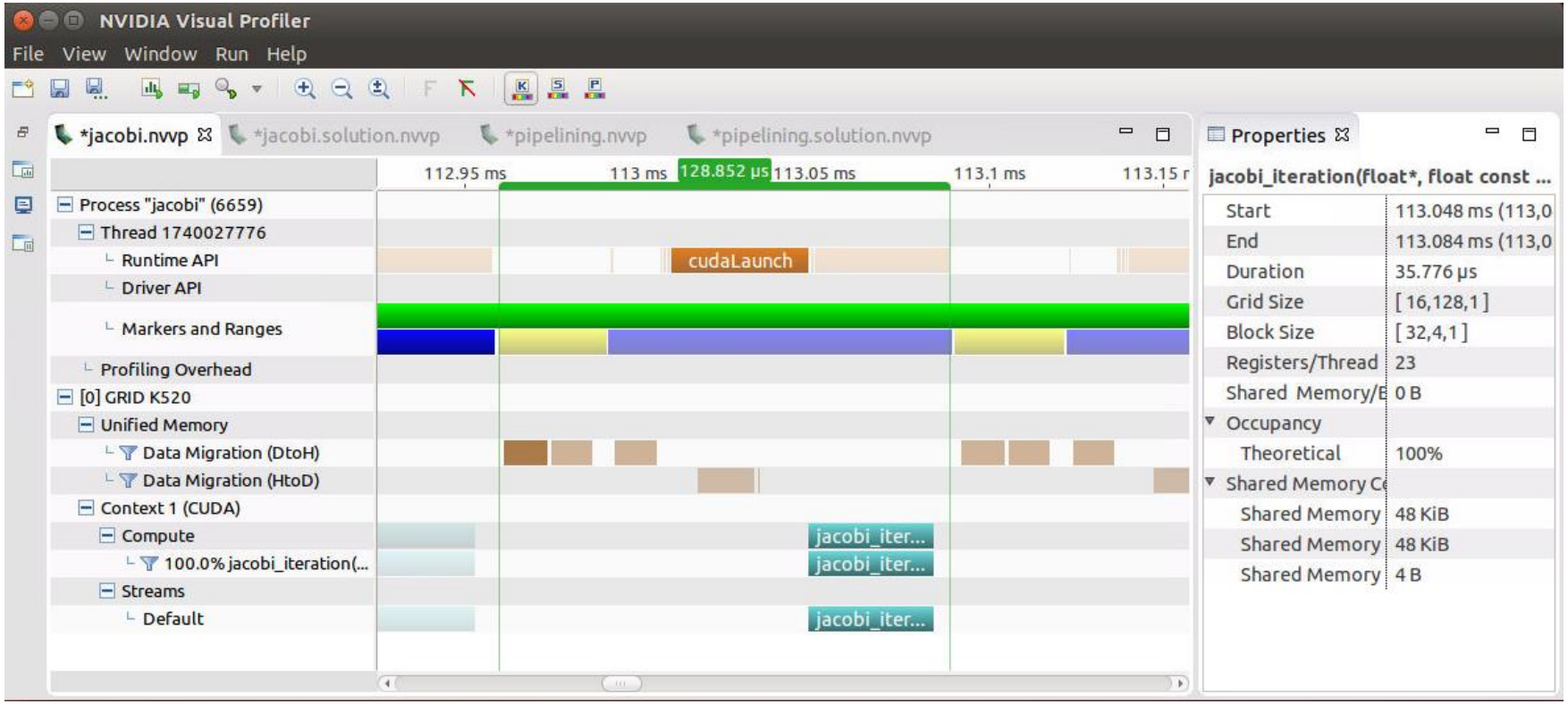
```
-bash-4.2$ export CUDA_MANAGED_FORCE_DEVICE_ALLOC=1
-bash-4.2$ make
nvcc -DUSE_NVTX -arch=sm_37 jacobi.cu -lnvToolsExt -o jacobi
./Jacobi
0
[...]
1000
Runtime 0.111733 seconds.
```

Make Targets:

`run:`	run `jacobi` (default)
`jacobi:`	build `jacobi` binary
`memcheck:`	run with `cuda-memcheck`
`profile:`	profile with `nvprof`

JACOBI

Initial Version



JACOBI

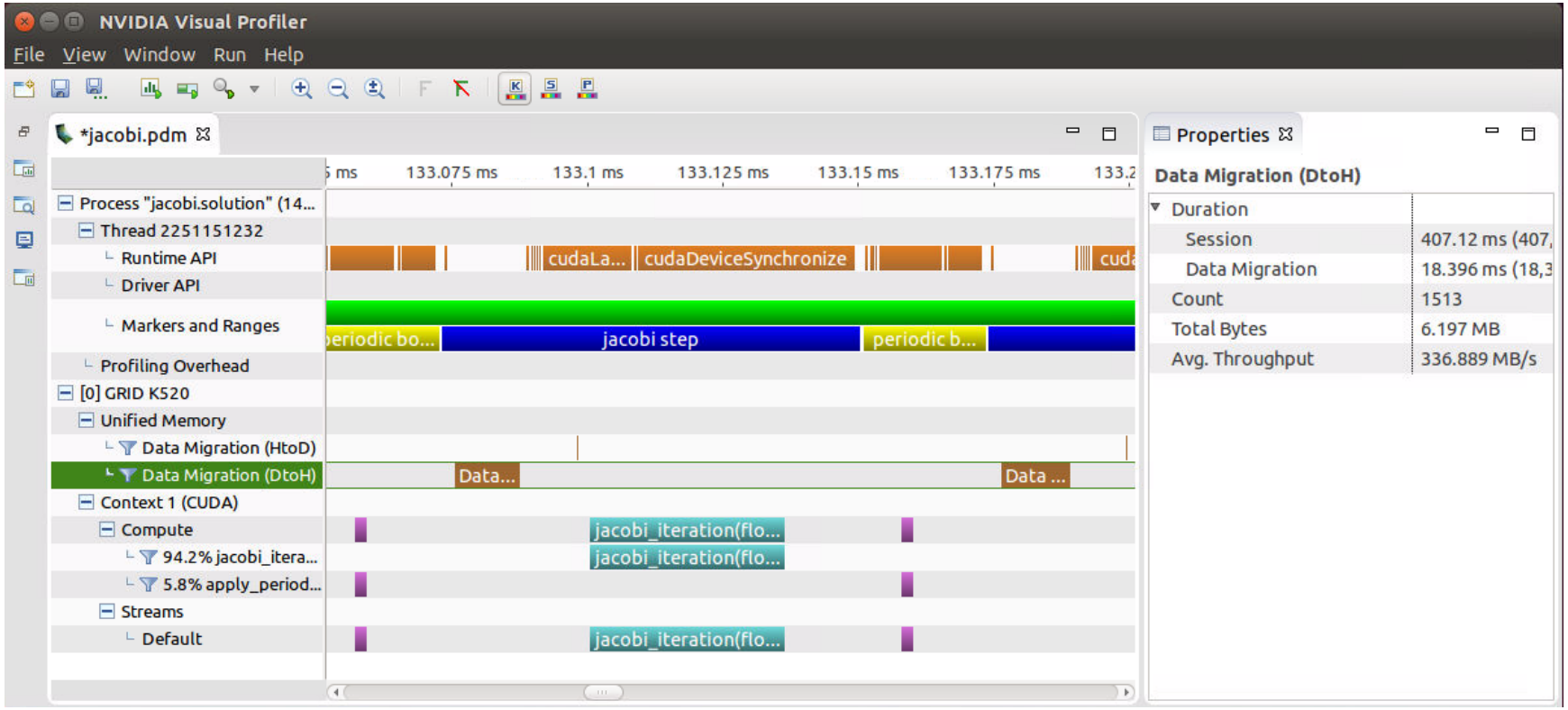
Apply boundary conditions GPU kernel

```
PUSH_RANGE("periodic boundary conditions",2)
//Apply periodic boundary conditions
apply_periodic_bc<<<dim3(nx/128),dim3(128)>>>(a,nx,ny);
CUDA_CALL(cudaGetLastError());
CUDA_CALL(cudaDeviceSynchronize());

POP_RANGE
```

See: https://devblogs.nvidia.com/parallelforall/cuda-pro-tip-generate-custom-application-profile-timelines-nvtx/ for details on PUSH_RANGE and POP_RANGE macros.

JACOBI

Apply boundary conditions GPU kernel



JACOBI

```
PUSH_RANGE("jacobi step",1)
jacobi_iteration<<<dim3(nx/32,ny/4),dim3(32,4)>>>(
                                  a_new,a,
                                  nx,ny,
                                  weights[0]);
CUDA_CALL(cudaGetLastError());
CUDA_CALL(cudaDeviceSynchronize());
POP_RANGE
```

CPU reading **weights[0]** before launch triggers DtoH Data Migration

See: https://devblogs.nvidia.com/parallelforall/cuda-pro-tip-generate-custom-application-profile-timelines-nvtx/ for details on PUSH_RANGE and POP_RANGE macros.

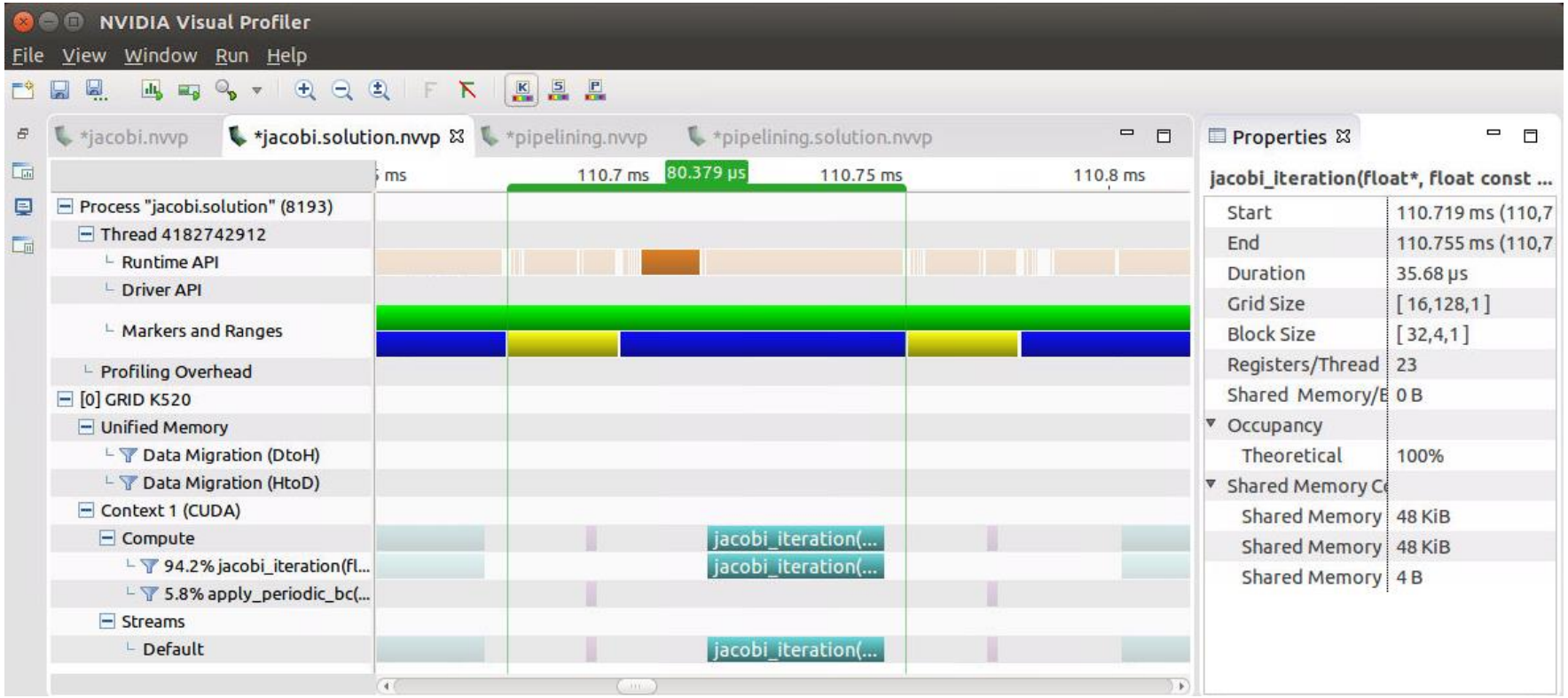
JACOBI

Solution

```
const float weight = weights[0];
while ( iter <= iter_max ) {
      jacobi_iteration<<<dim3(nx/32,ny/4),dim3(32,4)>>>(a_new,a,nx,ny,weight);
      CUDA_CALL(cudaGetLastError());
      CUDA_CALL(cudaDeviceSynchronize());

      std::swap(a,a_new);
      //Apply periodic boundary conditions
      apply_periodic_bc<<<dim3(nx/128),dim3(128)>>>(a,nx,ny);
      CUDA_CALL(cudaGetLastError());
      CUDA_CALL(cudaDeviceSynchronize());
      if ( 0 == iter%100 )
                  std::cout<<iter<<std::endl; iter++;
}
```

JACOBI

Solution



JACOBI

Results

	RUNTIME (S)	HOST TO DEVICE - DATA MIGRATIONS*	DEVICE TO HOST - DATA MIGRATIONS*
Initial	0.112	982.334 MB	13.73047 MB
Final	0.040	2.003906 MB	2.003906 MB

Optional Task: cudaDeviceSynchronize() calls within while loop are no longer necessary and can be removed for further speedup

*nvprof --unified-memory-profiling per-process-device ./jacobi