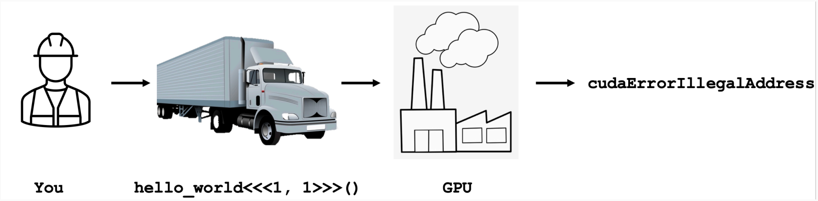
Transparent Asynchronous Compute Made Easy With PETSc

Jacob Faibussowitsch

June 6, 2023

Your GPU Code Is Slow



GPUs are like a factory:

- → Big startup cost
- → Needs steady supply of work
- → Works best in bulk

Difficult to keep saturated for small jobs

- → Performance left on the table
- → You paid for the whole GPU, you should use the whole GPU

A Simple Example

```
PetscReal norm;

// Must copy result D2H and synchronize
VecNorm(x, NORM_2, &norm);
norm = 1.0 / norm;
// Must copy norm H2D and synchronize after
VecScale(x, norm);
```

This is a common scenario!

- Functions operate on, or produce scalar values
- Values piped to next GPU function after basic manipulation
- Results are immediate $\rightarrow$ must synchronize GPU after each call

A Simple Example

```
PetscReal norm;

// Must copy result D2H and synchronize
VecNorm(x, NORM_2, &norm);
norm = 1.0 / norm;
// Must copy norm H2D and synchronize after
VecScale(x, norm);
```

This is a common scenario!

- Functions operate on, or produce scalar values
- Values piped to next GPU function after basic manipulation
- Results are immediate $\rightarrow$ must synchronize GPU after each call

Ideally this is all done in a stream on the GPU...

What Are GPU Streams?

```
typedef pthread_t gpu_stream_t;
```

Essentially threads, both the good and the bad

- ↑ Putting work "on" a stream → launching a thread
- ↑ Efficient "communication" via recorded events/semaphores

- ↓ Non-linear execution path, hard to grok
- ↓ Race condition hazards
- ↓ Deadlock hazards

What Are GPU Streams?

```
typedef pthread_t gpu_stream_t;
```

Essentially threads, both the good and the bad

- ↑ Putting work “on” a stream → launching a thread
- ↑ Efficient “communication” via recorded events/semaphores

- ↓ Non-linear execution path, hard to grok
- ↓ Race condition hazards
- ↓ Deadlock hazards

Surely every vendor has agreed on a single implementation… right?

One Wrapper To Rule Them All

"*Just use a* `cudaStream_t`"

\- NVIDIA

"*Just use a* `hipStream_t`"

\- AMD

"*Just use a* `sycl::queue`"

\- Intel

One Wrapper To Rule Them All

`PetscDeviceContext`

1. Abstract away vendor-specific data structures
2. "Do the right thing" for missing functionality
3. Give high-level control over streams and synchronization primitives

One Wrapper To Rule Them All

`PetscDeviceContext`

Bottom Line

Dictate how, when, and where, a particular GPU op runs

Petsc::ManagedMemory: The Solution To All Your Problems

```
Petsc::ManagedReal norm;
PetscDeviceContext dctx;

// Retrieve the current active device context (or create one)
PetscDeviceContextGetCurrentContext(&dctx);
// Store result in device memory (asynchronously!)
VecNormAsync(x, NORM_2, &norm, dctx);
// Evaluate the expression on device (asynchronously!)
norm = eval(dctx, 1.0 / norm);
// Use result (asynchronously!)
VecScaleAsync(x, norm, dctx);
// Wait for results to be ready
PetscDeviceContextSynchronize(dctx);
```

Petsc::ManagedMemory: The Solution To All Your Problems

```
Petsc::ManagedReal norm;
PetscDeviceContext dctx;

// Retrieve the current active device context (or create one)
PetscDeviceContextGetCurrentContext(&dctx);
// Store result in device memory (asynchronously!)
VecNormAsync(x, NORM_2, &norm, dctx);
// Evaluate the expression on device (asynchronously!)
norm = eval(dctx, 1.0 / norm);
// Use result (asynchronously!)
VecScaleAsync(x, norm, dctx);
// Wait for results to be ready
PetscDeviceContextSynchronize(dctx);
```

✓ Values computed in device memory

Petsc::ManagedMemory: The Solution To All Your Problems

```
Petsc::ManagedReal norm;
PetscDeviceContext dctx;

// Retrieve the current active device context (or create one)
PetscDeviceContextGetCurrentContext(&dctx);
// Store result in device memory (asynchronously!)
VecNormAsync(x, NORM_2, &norm, dctx);
// Evaluate the expression on device (asynchronously!)
norm = eval(dctx, 1.0 / norm);
// Use result (asynchronously!)
VecScaleAsync(x, norm, dctx);
// Wait for results to be ready
PetscDeviceContextSynchronize(dctx);
```

- ✓ Values computed in device memory
- ✓ Support for arbitrary expressions $\rightarrow$ values *stay* on device

Petsc::ManagedMemory: The Solution To All Your Problems

```
Petsc::ManagedReal norm;
PetscDeviceContext dctx;

// Retrieve the current active device context (or create one)
PetscDeviceContextGetCurrentContext(&dctx);
// Store result in device memory (asynchronously!)
VecNormAsync(x, NORM_2, &norm, dctx);
// Evaluate the expression on device (asynchronously!)
norm = eval(dctx, 1.0 / norm);
// Use result (asynchronously!)
VecScaleAsync(x, norm, dctx);
// Wait for results to be ready
PetscDeviceContextSynchronize(dctx);
```

- ✓ Values computed in device memory
- ✓ Support for arbitrary expressions → values *stay* on device
- ✓ Ability to await results → functions may be asynchronous

The Rubber Hits The Road

```
Petsc::ManagedReal  norm;
PetscScalar        *cpu_array;
PetscDeviceContext  dctx_a, dctx_b, dctx_c;

// These are all *separate* streams
PetscDeviceContextGetCurrentContext(&dctx_a);
PetscDeviceContextDuplicate(dctx_a, &dctx_b);
PetscDeviceContextDuplicate(dctx_a, &dctx_c);
// Store result in device memory (asynchronously!)
VecNormAsync(x, NORM_2, &norm, dctx_a);
// Evaluate the expression on device (asynchronously!)
norm = eval(dctx_b, 1.0 / norm);
// Use result (asynchronously!)
VecScaleAsync(x, norm, dctx_c);
// Get results (...synchronously?)
VecGetArray(x, &cpu_array);
```

To Go Even Further Beyond

```
Petsc::ManagedReal  norm;
PetscScalar        *cpu_array;
PetscDeviceContext  dctx_a, dctx_b, dctx_c;

// These are all *separate* streams
PetscDeviceContextGetCurrentContext(&dctx_a);
PetscDeviceContextDuplicate(dctx_a, &dctx_b);
PetscDeviceContextDuplicate(dctx_a, &dctx_c);
// Store result in device memory (asynchronously!)
VecNormAsync(x, NORM_2, &norm, dctx_a);
// Evaluate the expression on device (asynchronously!)
norm = eval(dctx_b, 1.0 / norm);
// Use result (asynchronously!)
VecScaleAsync(x, norm, dctx_c);
// Get results (synchronously!)
VecGetArray(x, &cpu_array);
// PetscDeviceContextSynchronize(dctx); ???
```

To Go Even Further Beyond

```
Petsc::ManagedReal  norm;
PetscScalar        *cpu_array;
PetscDeviceContext  dctx_a, dctx_b, dctx_c;

// These are all *separate* streams
PetscDeviceContextGetCurrentContext(&dctx_a);
PetscDeviceContextDuplicate(dctx_a, &dctx_b);
PetscDeviceContextDuplicate(dctx_a, &dctx_c);
// Store result in device memory (asynchronously!)
VecNormAsync(x, NORM_2, &norm, dctx_a);
// Evaluate the expression on device (asynchronously!)
norm = eval(dctx_b, 1.0 / norm);
// Use result (asynchronously!)
VecScaleAsync(x, norm, dctx_c);
// Get results (synchronously!)
VecGetArray(x, &cpu_array);
// PetscDeviceContextSynchronize(dctx); ???
```

- ✓ Automatically serializes stream dependencies for you
- ✓ Even across ”regular” API

To Go Even Further Beyond

```
Petsc::ManagedReal  norm;
PetscScalar        *cpu_array;
PetscDeviceContext  dctx_a, dctx_b, dctx_c;

// These are all *separate* streams
PetscDeviceContextGetCurrentContext(&dctx_a);
PetscDeviceContextDuplicate(dctx_a, &dctx_b);
PetscDeviceContextDuplicate(dctx_a, &dctx_c);
// Store result in device memory (asynchronously!)
VecNormAsync(x, NORM_2, &norm, dctx_a);
// Evaluate the expression on device (asynchronously!)
norm = eval(dctx_b, 1.0 / norm);
// Use result (asynchronously!)
VecScaleAsync(x, norm, dctx_c);
// Get results (synchronously!)
VecGetArray(x, &cpu_array);
// PetscDeviceContextSynchronize(dctx); ???
```

- ✓ Automatically serializes stream dependencies for you
- ✓ Even across ”regular” API !!!

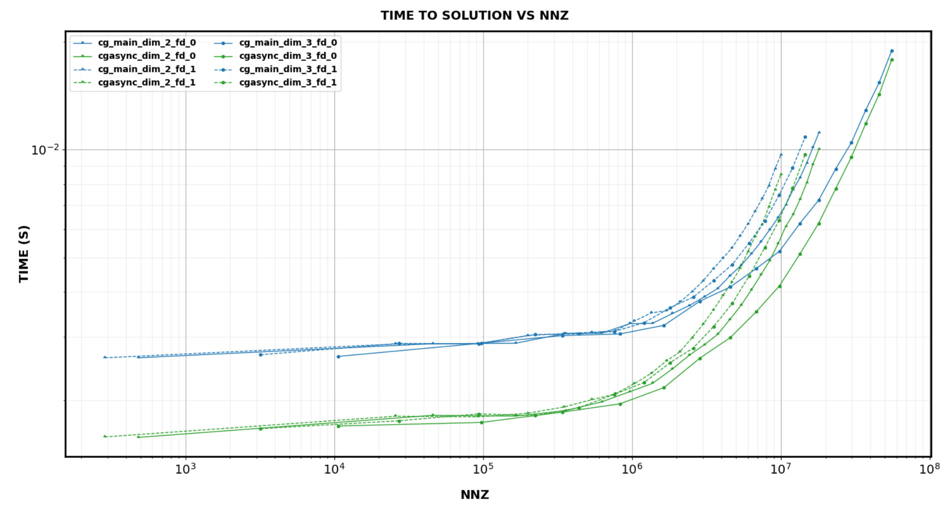
Cool... But How Does It Scale?

CG Benchmark:

- Run on ANL Polaris:
  - GPU: NVIDIA A100
  - CPU: AMD EPYC “Milan”
- Solve Laplace equation of varying size and density[1]
- 20 KSP iterations (for simplicity of comparison)
- Jacobi preconditioning
- 1 MPI Rank

[1] Cartesian product of 2D, 3D, with, and without fnite diference stencil

Time To Solution



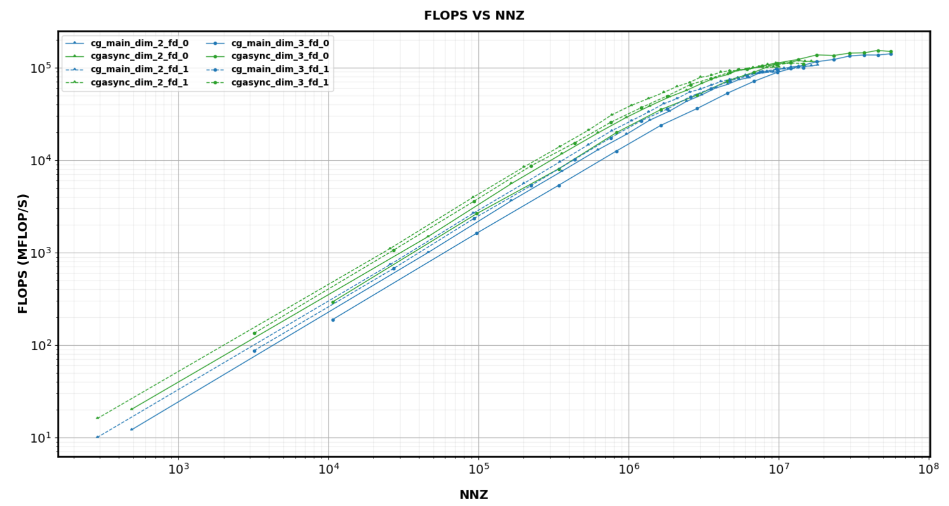
`cg_main`:

- `-ksp_type cg` on `main` branch
- Synchronous

`cgasync`:

- `-ksp_type cgasync` on exp. branch
- Fully Asynchronous

Flops



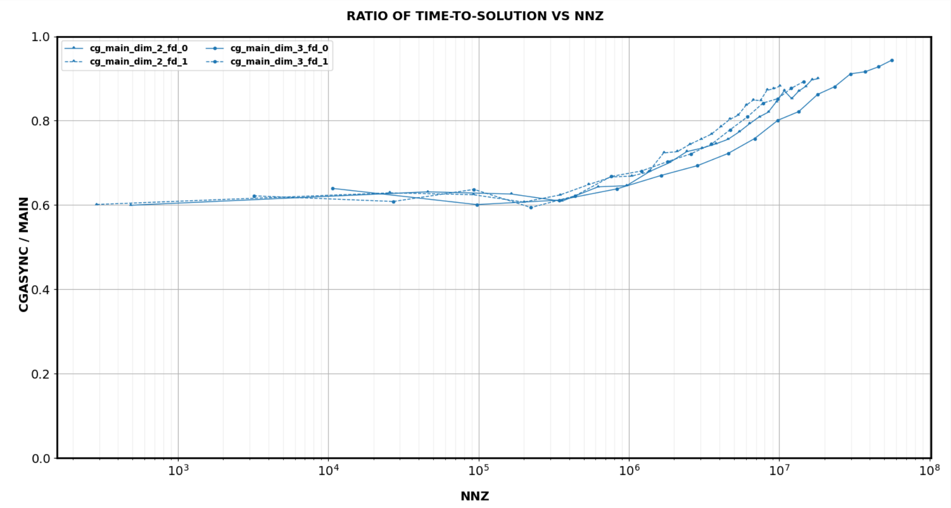
`cg_main`:

- `-ksp_type cg` on `main` branch
- Synchronous

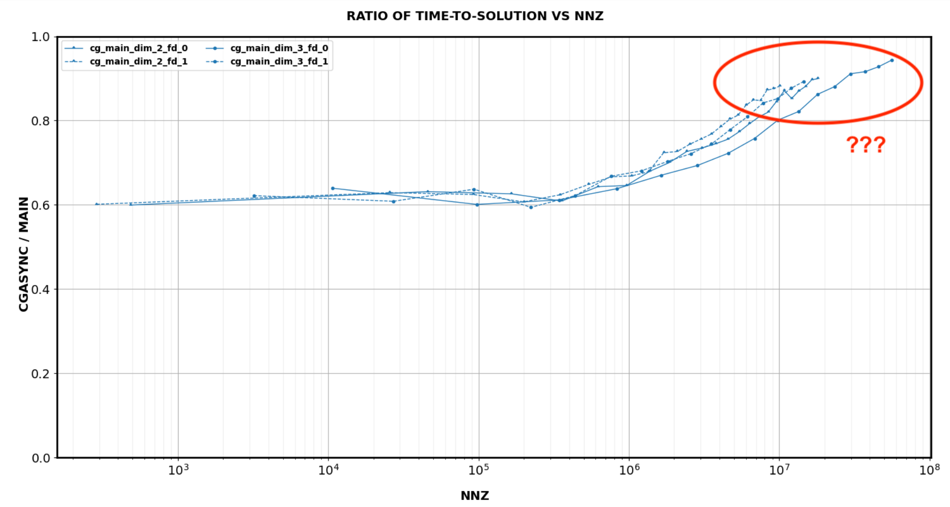
`cgasync`:

- `-ksp_type cgasync` on exp. branch
- Fully Asynchronous

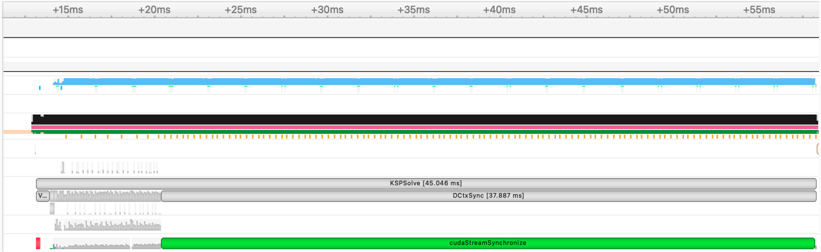
Time To Solution Ratio



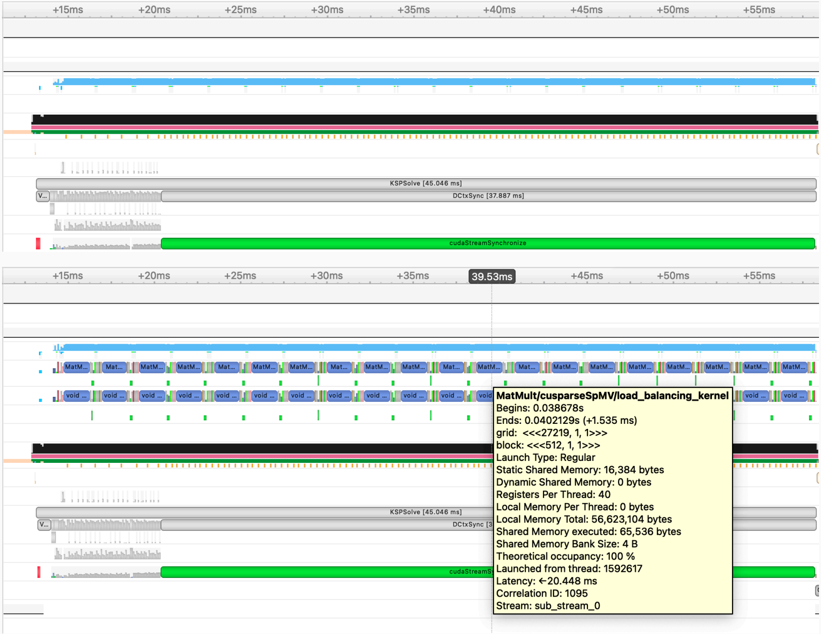
Time To Solution Ratio



Mystery Solved?



Mystery Solved?



Conclusion

Key Takeaways:

- ✓ Faster across the board
- ✓ Much faster ($\sim 1.8\times$) if latency bound
- ✓ Speedup diminishes as solve becomes “work bound”...

Conclusion

Key Takeaways:

- ✓ Faster across the board
- ✓ Much faster ($\sim 1.8\times$) if latency bound
- ✓ Speedup diminishes as solve becomes "work bound"...
- ✓ ...but that assumes you cannot fill the time with other work!

Future Work

- ✓ GMRES implementation (finishing touches)
- ✓ Automatic runtime kernel fusion
  - Implemented for `Vec`, but benefits still under investigation...
  - Needs tight integration with `Mat` to be truly useful
- ✓ Ability to cancel GPU work in-flight (“unlaunch” a kernel)

Future Work

- ✓ GMRES implementation (finishing touches)
- ✓ Automatic runtime kernel fusion
  - Implemented for `Vec`, but benefits still under investigation...
  - Needs tight integration with `Mat` to be truly useful
- ✓ Ability to cancel GPU work in-flight ("unlaunch" a kernel)

Thanks For Listening!

Check out the branch

`jacobf/2022-11-28/petsc-managed-memory`[a]

Any questions?

[a] *https://gitlab.com/petsc/petsc/-/merge_requests/6178*