



Schrödinger

S61156: Accelerating Drug Discovery: Optimize Dynamic GPU Workflows with CUDA Graphs, Mapped Memory, C++ Coroutines, and More

Jiqun Tu, Senior Developer Technology Engineer, NVIDIA

Ellery Russell, Tech Lead - Desmond Engine, Schrodinger, Inc.

Structure of This Talk

Real world case study on optimizing an already mature, fully GPU-resident algorithm to achieve high performance on today's ever-growing GPU cards

- Strategies considered and choices made
- High-quality, maintainable software

Contents

- Domain overview - Molecular Dynamics (MD), Free Energy Perturbation (FEP)
- Applying CUDA Graphs
- De-risking stream-recorded CUDA graphs
- Coroutines for additional in-process parallelism
- Kernel optimizations - shared memory
- Future strategies

Molecular Dynamics and Desmond Overview

Molecular Dynamics Overview

A classical physics representation of chemistry

- Uses a ‘forcefield’ to approximate the true quantum interactions
- System is described by the position and momenta of atoms (or more generic particles) that comprise that system
- Simulates the time evolution of the system according to Newton’s equations of motion, producing a trajectory of all the particles

MD simulations sample states from the Boltzmann distribution

- This property allows us to calculate statistical averages, which are often of more interest than the specific trajectory

$$\pi(x) = \frac{e^{-U_{\text{total}}(x)/k_B T}}{Z(N, V, T)}$$

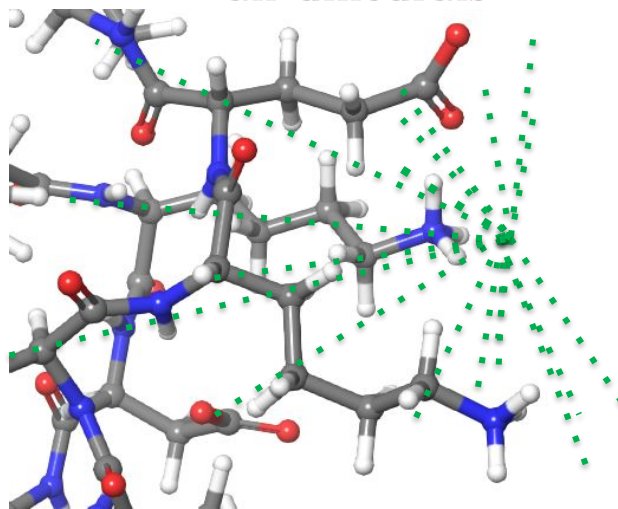
The Boltzmann factor/weight

The partition function, or “sum over states”

Molecular Dynamics Overview

The forcefield is used to calculate the total potential energy of each atom:

$$U_{\text{total}} = \sum_{\text{all bonds}} U_{\text{bond}} + \sum_{\text{all angles}} U_{\text{angle}} + \sum_{\text{all dihedrals}} U_{\text{dihedral}} + \sum_{\text{all pairs}} U_{\text{electro}} + \sum_{\text{all pairs}} U_{\text{LJ}}$$

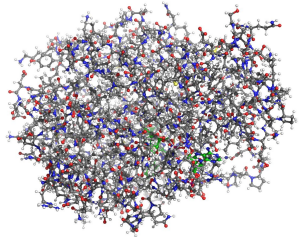


$$\text{Force} = -\nabla U_{\text{total}}$$

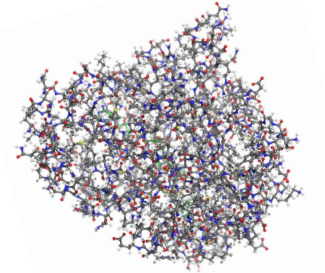
The gradient of the potential energy with respect to position gives the force acting on each atom

Molecular Dynamics Overview

Starting structure



Final structure



For a very small "time-step"

Initialize

Calculate forces

Update positions
and velocities

Thermodynamic parameters
e.g temperature and pressure

Introducing Desmond

- ‘Desmond’ is Schrodinger’s Molecular Dynamics Engine
 - Originally developed by DE Shaw Research
 - Schrodinger now provides commercial licenses and maintains a very active fork
- A high-performance MD Engine
 - Purpose built GPU port was introduced in 2014
 - Scalability, throughput, and scientific accuracy
 - GPU resident, no need to move data to CPU except for file output
- Impact
 - Schrodinger averages around **8,000-10,000 GPUs** in use at all times
 - Desmond, and its main application FEP+, is the primary solution in use by essentially all major pharmaceutical companies for computational binding affinity prediction
 - Binding affinity is the holy grail of computational drug discovery

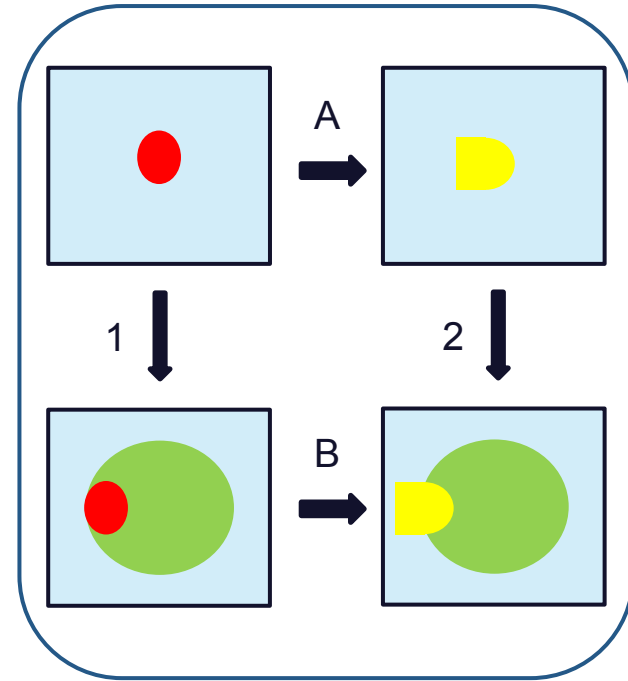
FEP+ Overview

Thermodynamic cycle

$$\Delta\Delta G = \Delta G_1 - \Delta G_2 = \Delta G_A - \Delta G_B$$

In this picture, 1 and 2 are difficult to simulate directly.

Instead, we simulate the A (solvent) and B (complex) legs of this thermodynamic cycle. We can then use knowledge of 1 to determine 2, or vice versa.



FEP+ Overview

FEP+ (our Free Energy Perturbation software) allows us to determine the binding free energy of a drug-like molecule to a target protein

Real starting state

V_0



$\lambda = 0$

G_0^0

Real final state

V_1



$\lambda = 1$

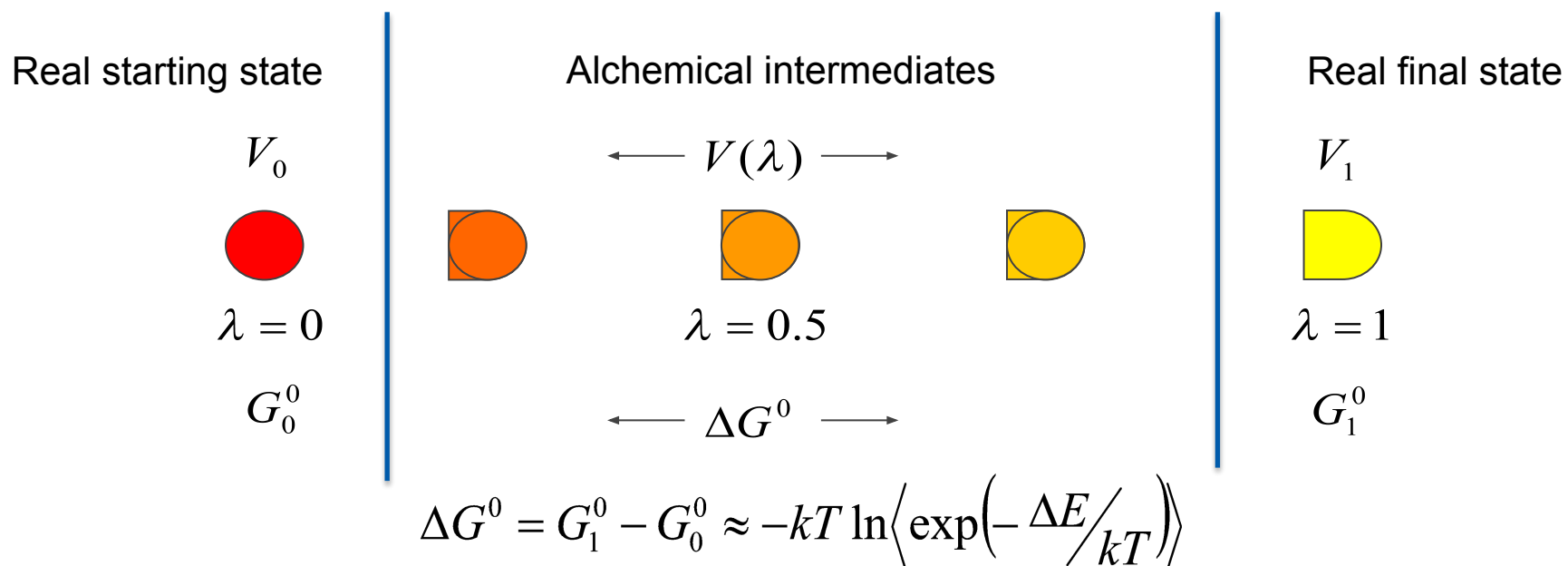
G_1^0

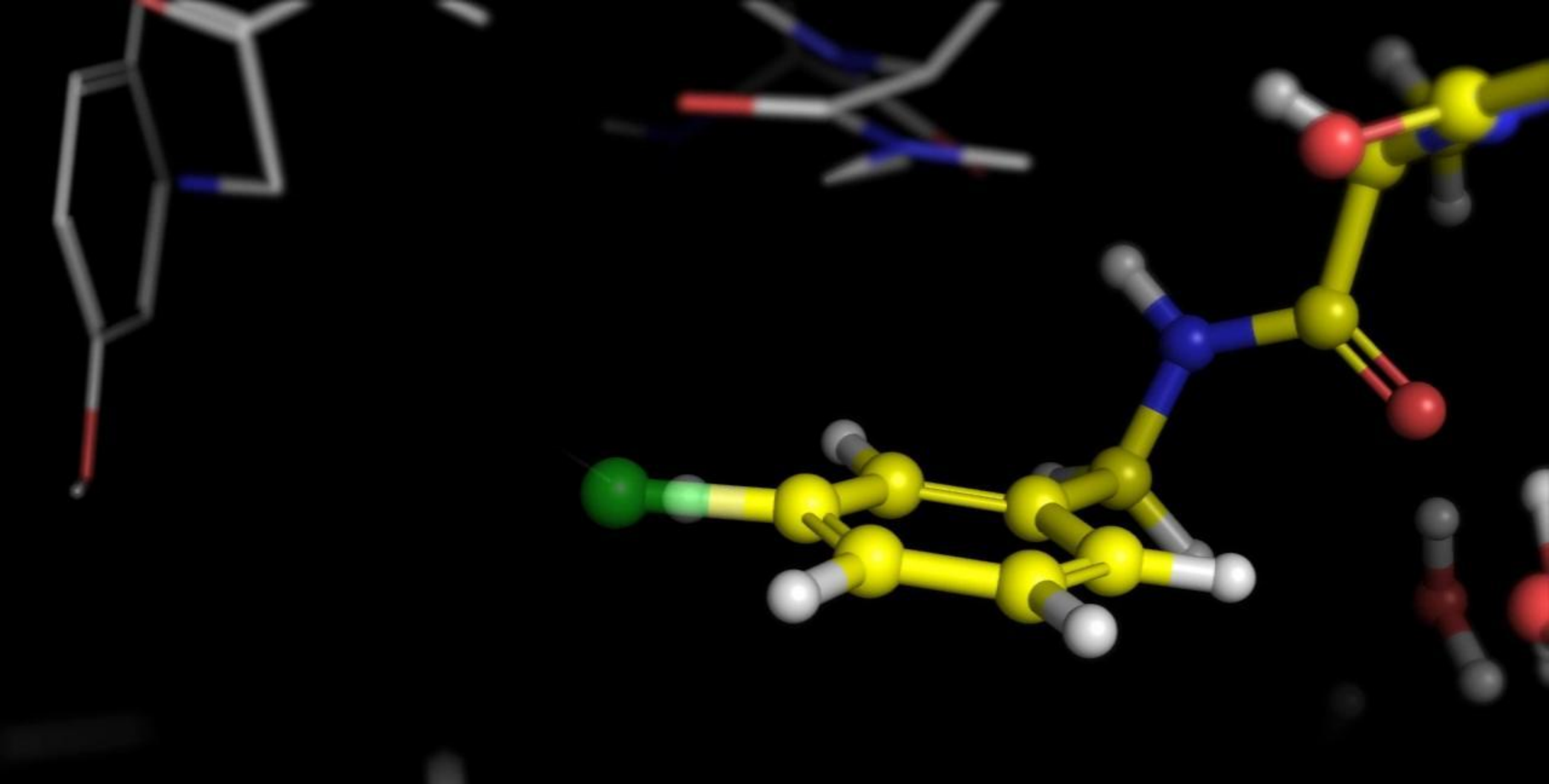
$$\Delta G^0 = G_1^0 - G_0^0 \approx -kT \ln \left\langle \exp \left(-\frac{\Delta E}{kT} \right) \right\rangle$$

FEP+ Overview

FEP+ (our Free Energy Perturbation software) allows us to determine the binding free energy of a drug-like molecule to a target protein

- This requires many concurrent MD simulations for “alchemical intermediates”
- Because we usually alchemically perturb a small subset of the system, all of the simulations are highly similar in compute requirements





Wall clock time: 5.91 hr
Simulation time: 3.69 ns

$$\Delta\Delta G_{\text{bind}} = -0.68 \text{ kcal/mol}$$

Desmond performance - Reduce serial bottlenecks

As GPUs continue to get larger, serial bottlenecks become more evident

- Latency of CUDA kernel launches begins to dominate
 - As we will discuss later, we can minimize these with CUDA graphs
 - Addressing this can ensure that at least some of the SMs are busy all the time
- There will still be times when not all SMs are busy due to serial dependencies on small kernels
 - This forms the lower limit for time to solution
 - In molecular dynamics, we expect alternating small and large kernels, as calculating forces can be $O(N \log N)$ or $O(K * N)$, while updating positions and velocities is $O(N)$.

Strong Scaling and Amdahl's Law

- These are classic strong scaling issues:
 - Strong scaling:
 - “Solution time as processors are increased and problem size is constant”
 - Amdahl's Law: $S_{\text{latency}}(s) = \frac{1}{(1-p) + \frac{p}{s}}$
 - Speedup is limited by the part of the task which is not parallelizable

$$\begin{cases} S_{\text{latency}}(s) \leq \frac{1}{1-p} \\ \lim_{s \rightarrow \infty} S_{\text{latency}}(s) = \frac{1}{1-p} \end{cases}$$

- S_{latency} is the theoretical speedup of the execution of the whole task;
- s is the speedup of the part of the task that benefits from improved system resources;
- p is the proportion of execution time that the part benefiting from improved resources originally occupied.

https://en.wikipedia.org/wiki/Amdahl%27s_Law

Desmond performance - Focus on Throughput

- Time to solution may be the most 'attractive' target, but throughput is often more important - this is the case at Schrodinger
 - Minimum time to solution is the order of hours - “fast enough” for drug discovery
- Schrodinger averages around **8,000-10,000 GPUs** in use at all times
 - Customer usage is even more
 - Current largest supercomputer “Frontier” has ~37,000 GPUs (albeit larger GPUs)
 - Our workload is comparable to constantly running a “medium” supercomputer
- Throughput Strategy:
 - We can “cover up” serial bottlenecks in our program by running many independent simulations on the same GPU

Desmond performance - Summary

- So now we know the problems we're facing
 - Fixed-latency CUDA API calls
 - Serial dependencies on small kernels
 - More generally, Amdahl's Law
- But, we have one thing going for us
 - Focusing on throughput simplifies the problem, we can attempt to overlap multiple simulations onto a single GPU
 - By expanding the problem, we make it more parallelizable
- If we achieve ~100% utilization - is there any room for improvement in critical kernels?

- In the following slides, you'll see our approach to solving these problems with the tools available in CUDA
 - And the speedups we're able to achieve...

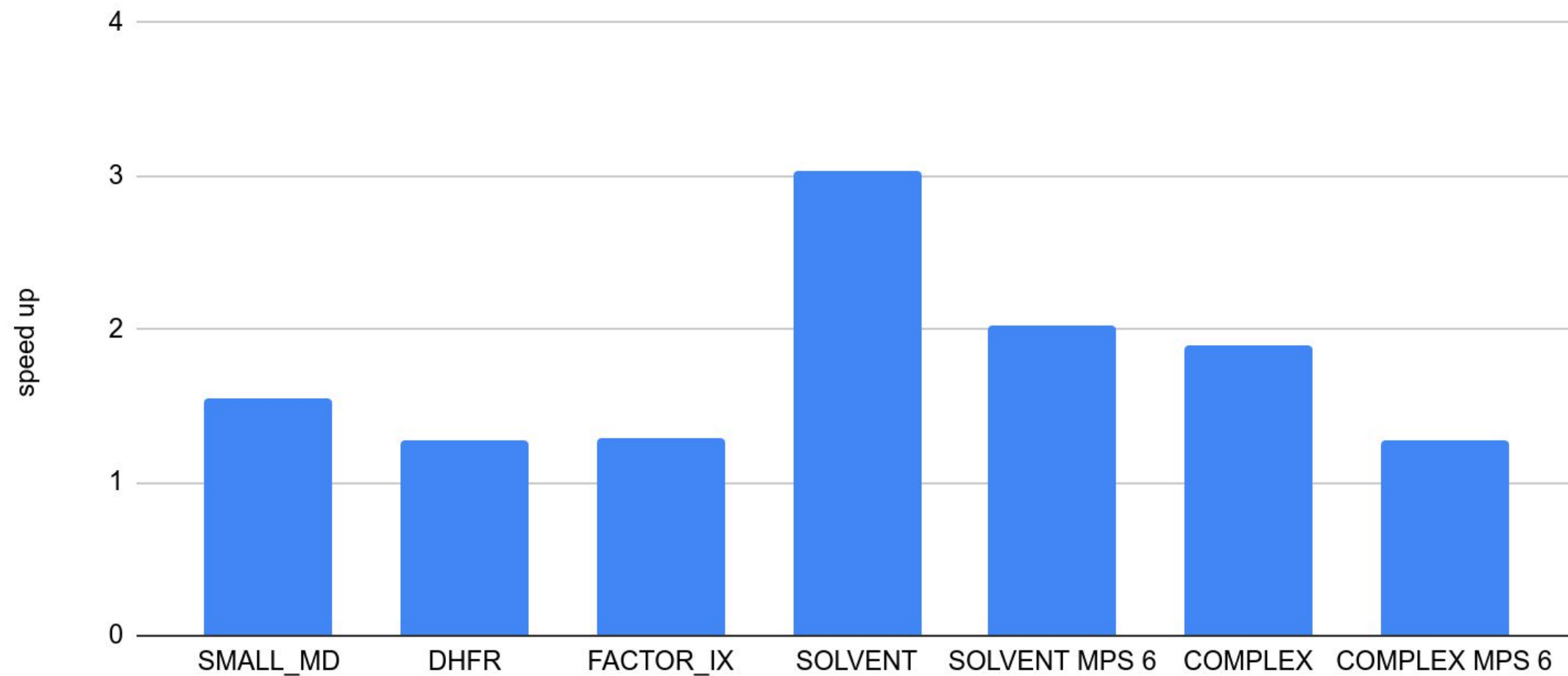
Performance Improvement

- Benchmarked on an A100-40GB

ns/day	# of atoms	2020-4	latest	speed up
SMALL_MD	4k	891	1381	1.55
DHFR	23k	659	843	1.28
FACTOR_IX	90k	287	371	1.29
SOLVENT	5k x 12	100	304	3.04
SOLVENT MPS 6	5k x 12	447	905	2.02
COMPLEX	35k x 12	71	134	1.89
COMPLEX MPS 6	35k x 12	159	204	1.28

Performance Improvement

speed latest versus 2020-4



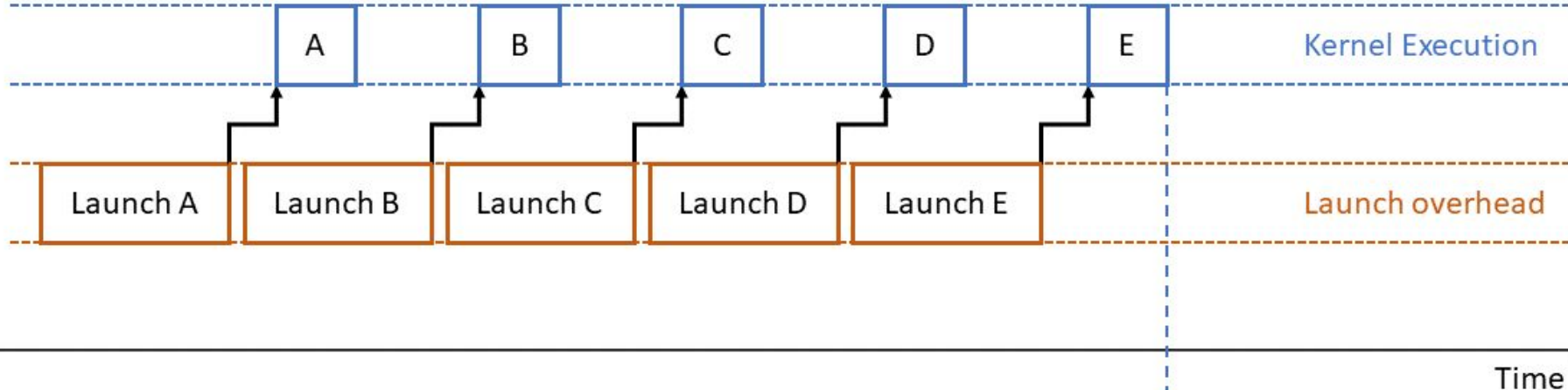
CUDA Graph

CUDA Graph

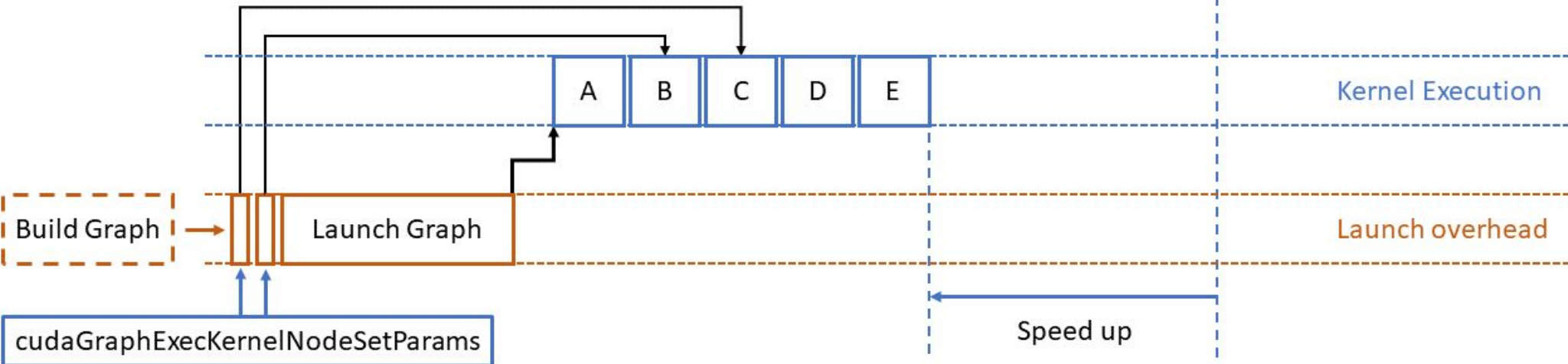
- CUDA Graph: group the kernels and CUDA APIs together into a graph and execute them according to a dependency tree.
- Benefits:
 - Less total overhead
 - Faster overall kernel/API performance, e.g., less gaps between the kernels
 - dependency is handled directly (instead of being specified by the user with CUDA streams/events)
- But ... one needs to construct the graph beforehand.

CUDA Graph

Without Graph

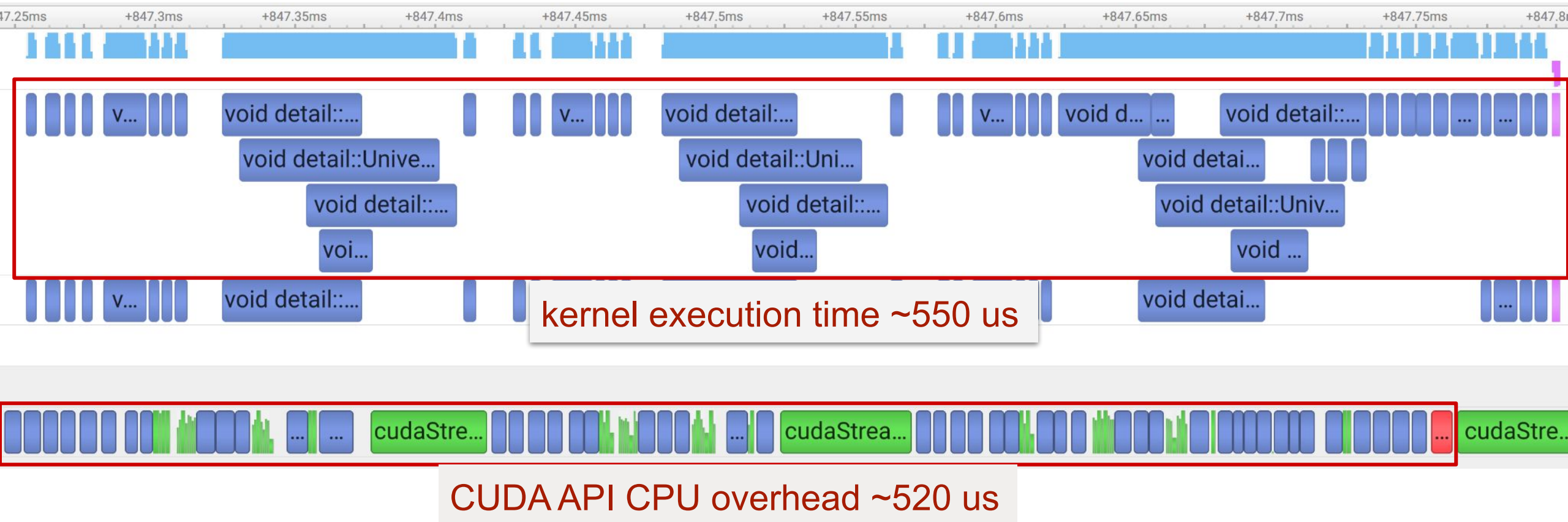


With Graph



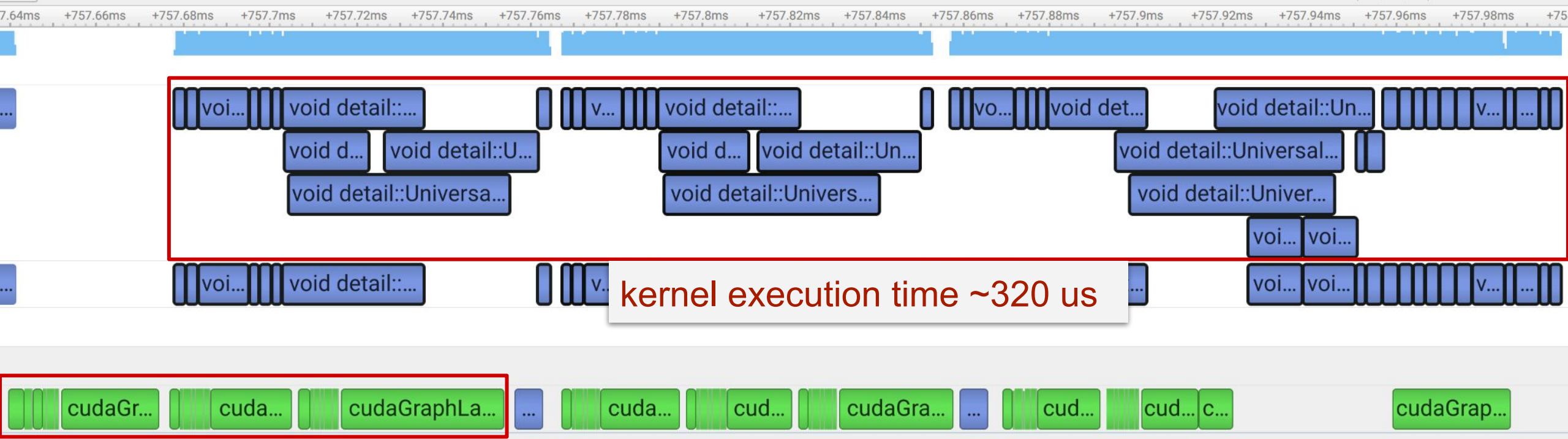
CUDA Graph

- Screenshot of a DHFR **without** the CUDA graph changes.



CUDA Graph

- Screenshot of a DHFR **with** the CUDA graph changes.



CUDA API CPU overhead ~115 us

CUDA Graph

- Two ways to construct a CUDA graph.
- **Stream capture:**
 - All kernels/APIs launched to the stream are captured and a graph with the captured nodes are return after the capture.
 - Dependencies are figured out automatically based on the CUDA events (if any).
 - The kernel/API parameters are recorded by **value**.

```
cudaStreamBeginCapture(stream, cudaStreamCaptureModeGlobal);
```

```
// Launch kernels, CUDA APIs to stream
```

```
// End the capture and instantiate the graph
```

```
cudaGraph_t _captured_graph;
```

```
cudaStreamEndCapture(stream, &_captured_graph);
```

```
cudaGraphInstantiate(&_graph_exec, _captured_graph, nullptr, nullptr, 0);
```

```
// Launch the graph
```

```
cudaGraphLaunch(_graph_exec, some_other_stream);
```

CUDA Graph

- Two ways to construct a CUDA graph.
- **Explicit API:**
 - Nodes are added to the graph explicitly.
 - Node dependencies are specified explicitly.

```
// Manually add a new kernel node
cudaGraphNode_t new_node;
cudaKernelNodeParams _params_cuda;
cudaGraphAddKernelNode(&new_node, _manual_graph, _deps, _dep_count, &_params_cuda);

cudaGraphInstantiate(&_graph_exec, _manual_graph, nullptr, nullptr, 0);
// Launch the graph
cudaGraphLaunch(_graph_exec, some_other_stream);
```


CUDA Graph

Deal with kernels with dynamic parameters:

- In most cases we want to use the stream capture approach such that we do not need to rewrite the code completely.
- For a small number of kernels we want to have access to the kernel node handles such that we can update those kernel nodes at runtime, without needing to re-do the capture-then-instantiate process.
- Specifically, we have a kernel whose launch configurations change from invocation to invocation.

```
cudaStreamBeginCapture(stream, cudaStreamCaptureModeGlobal);
// Launch kernels, CUDA APIs to stream
// Get the current stream capturing graph
// ...
cudaStreamGetCaptureInfo_v2(stream, &capture_status, nullptr &_capturing_graph, &_deps, &_dep_count);
// Manually add a new kernel node and store the new node for future references
// ...
cudaGraphAddKernelNode(&new_node, _capturing_graph, _deps, _dep_count, &dynamic_params_cuda);
// ...
// Update the stream dependencies
cudaStreamUpdateCaptureDependencies(stream, &new_node, 1, 1);
// End the capture and instantiate the graph
cudaGraph_t _captured_graph;
cudaStreamEndCapture(stream, &_captured_graph);
cudaGraphInstantiate(&_graph_exec, _captured_graph, nullptr, nullptr, 0);
```

CUDA Graph

Deal with kernels with dynamic parameters:

- <https://developer.nvidia.com/blog/constructing-cuda-graphs-with-dynamic-parameters/>
- Combine the stream capture and explicit API method to be able to dynamically updating specific kernel node parameters (i.e., kernel launch configurations and kernel parameters) without launching every kernel with explicit API.
- While doing a stream capture, manually adding kernel nodes whose handles can be used later to update their parameters:

Graph comparison

Graph Recording introduces risk

- CUDA Graph stream recording, while convenient, introduces software risk
 - In order to use the recorded graph in a subsequent timestep, we require knowledge that the same sequence of CUDA operations will be performed
 - The recording/replaying layer must have absolute knowledge of all CUDA operations that will be performed by all code below it in the call stack
- Violates a number of tenets of software design by coupling the lowest levels to the highest, breaking encapsulation and separation of concerns.
 - Any future change to the lower level code must consider whether the graph recording layer needs to be made aware of it
 - Any incongruity between the recording code would be silently ignored
- How can we get the substantial benefit of using CUDA Graphs without sacrificing on software reliability and maintainability?

Graph Recording introduces risk - example

- Let's say an engineer wants to add the following behavior inside a function covered by CUDA graph recording
 - As a result of a 50/50 coin flip, one of the force evaluation kernels will either:
 - i. Not be evaluated - ``F = 0``
 - ii. Be evaluated with forces multiplied by 2 - ``F *= 2``
- A graph recorded on an arbitrary timestep either includes no kernel or the kernel multiplied by two
- At all subsequent timesteps the graph will replay the same operation, resulting in either no force, or double the force.
 - Silently incorrect results, could easily be missed
- For large software development teams, we can't expect all engineers to know about all parts of the code (or at least we really don't want to).

Graph Recording introduces risk - use explicit API

- Of course, we can use the explicit API to add nodes to the graph. This can potentially solve this software risk
 - For each kernel node, we record the local inputs that kernel node depends on into a key. These are aggregated into a key for the entire graph.
 - On any given timestep, we can call into the same stack to generate the graph keys, and if such a graph has already been instantiated, we can use it. Otherwise, we add a new graph.
 - For our example, we would have a key corresponding to the coinflip result, and two graphs associated with heads and tails results.
- Certainly robust, solves the aforementioned risk issues
- However, requires a large code investment. Essentially the entirety of the CUDA-aware codebase needs to be rewritten
 - Potential future option for Schrodinger, depending on the permanence of the feature.

Graph Recording solution - Graph comparison

- Idea: Periodically, on a timestep where we would 'replay' a graph, also re-record a new graph.
 - Then compare the two graphs for equality, down to the pointer and value parameters of CUDA kernels
- Intermediate solution between the safety of explicit graphs and the convenience of recorded graphs
 - We can choose the period to minimize the overhead, while still ensuring that graph comparisons happen hundreds of times during a simulation (which may have millions of timesteps)
 - Any comparison failure would cause the entire program to fail
- Significantly reduces the likelihood of uncaught errors, failures can be caught early by automated testing

Graph Comparison - technical challenges

- Unfortunately, things are not so simple
 - No CUDA API exists for comparing graphs
 - Graph Kernel nodes store their function pointers and parameters as type-erased ``void**``

```
bool compare_kernel_node_params(  
    const cudaKernelNodeParams& params_a,  
    const cudaKernelNodeParams& params_b) {  
    auto valueFields = [](const auto& params) {  
        return std::make_tuple(params.func,  
            params.gridDim, params.blockDim, ...);  
    };  
    bool vals_equal = valueFields(params_a) ==  
        valueFields(params_b);  
    bool kernel_params_equal = ???(params_a.kernelParams,  
        params_b.kernelParams)  
    return vals_equal && kernel_params_equal;  
}
```

```
struct __device_builtin__ cudaKernelNodeParams {  
    void* func; /**< Kernel to launch */  
    dim3 gridDim; /**< Grid dimensions */  
    dim3 blockDim; /**< Block dimensions */  
    void **kernelParams; /**< Array of pointers to kernel arguments*/  
    ...  
};
```

Values in ``kernelParams`` are copied internally in each recorded CUDA graph, so the pointers will not point to the same locations. Instead, we must compare the values pointed to.

However, we cannot safely compare them without their types. For 'trivial' types, we need at least the size of the type, and for non-trivial types we may need a custom `operator==` method.

Graph Comparison - technical challenges

- Some engineering will be required:
 - Create a global registry of ‘kernel comparators’, keyed by function pointers
 - For each recordable kernel, create a comparator that can cast the type-erased arguments back to their real types and compare them one by one.
 - When any kernel is called, register the comparator with the kernel’s function pointer (if not already)
 - When performing graph comparison, compare each node in the graph. If it is a kernel node, look up the registered kernel comparator and call it with the type-erased kernel arguments of the recorded node

Graph Comparison - Registration

```
class CompareArgsRegistry {
public:
    // Attempt to insert a function into the registry, generating a
    // corresponding comparator based on the function signature.
    template <typename... Args> void register(void(fcn)(Args...)) {

    }

    // Compares two sets of type-erased arguments to the same function pointer
    // by finding the registered comparator for that function.
    bool compare(void* func, void** args1, void** args2);

    static CompareArgsRegistry& instance();
private:
    using ComparatorFcn = std::function<bool(void**, void**)>;
    using RegistryT = std::map<void*, ComparatorFcn>;
    RegistryT m_registry{};
};
```

```
// generic functor kernel
template <typename FunctorT>
__global__ void generic_kernel(FunctorT functor, uint n)
{
    uint idx = threadIdx.x + blockDim.x * blockIdx.x;
    if (idx < n) {
        functor(idx);
    }
}

// arbitrary kernel functor
struct Truncate {
    float* data;
    float max_val;
    __device__ void operator()(uint idx) {
        data[idx] = min(data[idx], max_val);
    }
};

// typical registration and launch site
void truncate_buffer(GPUArray<float> arr, float max_val) {
    Truncate truncate_k{arr.data(), max_val};
    generic_kernel<<<...>>(truncate_k, arr.size());
}
```

Graph Comparison - Registration

```
class CompareArgsRegistry {
public:
    // Attempt to insert a function into the registry, generating a
    // corresponding comparator based on the function signature.
    template <typename... Args> void register(void(fcn)(Args...)) {
        auto erased_fcn = (void*)fcn;
        auto search = m_registry.find(erased_fcn);
        if (search == m_registry.end()) {
            m_registry[erased_fcn] = Comparator<Args...>;
        } else {
            // we can verify the registered function here, or simply no-op
        }
    }

    // Compares two sets of type-erased arguments to the same function pointer
    // by finding the registered comparator for that function.
    bool compare(void* func, void** args1, void** args2);

    static CompareArgsRegistry& instance();
private:
    using ComparatorFcn = std::function<bool(void**, void**)>;
    using RegistryT = std::map<void*, ComparatorFcn>;
    RegistryT m_registry{};
};
```

```
// generic functor kernel
template <typename FunctorT>
__global__ void generic_kernel(FunctorT functor, uint n)
{
    uint idx = threadIdx.x + blockDim.x * blockIdx.x;
    if (idx < n) {
        functor(idx);
    }
}

// arbitrary kernel functor
struct Truncate {
    float* data;
    float max_val;
    __device__ void operator()(uint idx) {
        data[idx] = min(data[idx], max_val);
    }
    // need to define equality operator for comparison
    bool operator==(const Truncate& other) {
        return data == other.data && max_val == other.max_val;
    }
};

// typical registration and launch site
void truncate_buffer(GPUArray<float> arr, float max_val) {
    Truncate truncate_k{arr.data(), max_val};
    CompareArgsRegistry::instance().register(generic_kernel<Truncate>);
    generic_kernel<<<...>>(truncate_k, arr.size());
}
```

Graph Comparison - Comparison

```
class CompareArgsRegistry {
public:
    // Attempt to insert a function into the registry, generating a
    // corresponding comparator based on the function signature.
    template <typename... Args> void register(void(fcn)(Args...)) {
        auto erased_fcn = (void*)fcn;
        auto search = m_registry.find(erased_fcn);
        if (search == m_registry.end()) {
            m_registry[erased_fcn] = Comparator<Args...>();
        } else {
            // we can verify the registered function here, or simply no-op
        }
    }

    // Compares two sets of type-erased arguments to the same function pointer
    // by finding the registered comparator for that function.
    bool compare(void* func, void** args1, void** args2);

    static CompareArgsRegistry& instance();
private:
    using ComparatorFcn = std::function<bool(void**, void**)>;
    using RegistryT = std::map<void*, ComparatorFcn>;
    RegistryT m_registry{};
};
```

```
bool CompareArgsRegistry::compare(void* func, void** args1, void** args2) {
    auto search = m_registry.find(func);
    if (search != m_registry.end()) {
        auto args_equal = search->second(args1, args2);
        return args_equal;
    }
    throw std::runtime_error("Comparator not found for function");
}
```

Graph Comparison - Comparison

```
class CompareArgsRegistry {
public:
    // Attempt to insert a function into the registry, generating a
    // corresponding comparator based on the function signature.
    template <typename... Args> void register(void(fcn)(Args...)) {
        auto erased_fcn = (void*)fcn;
        auto search = m_registry.find(erased_fcn);
        if (search == m_registry.end()) {
            m_registry[erased_fcn] = Comparator<Args...>();
        } else {
            // we can verify the registered function here, or simply no-op
        }
    }

    // Compares two sets of type-erased arguments to the same function pointer
    // by finding the registered comparator for that function.
    bool compare(void* func, void** args1, void** args2);

    static CompareArgsRegistry& instance();
private:
    using ComparatorFcn = std::function<bool(void**, void**)>;
    using RegistryT = std::map<void*, ComparatorFcn>;
    RegistryT m_registry{};
};
```

```
bool CompareArgsRegistry::compare(void* func, void** args1, void** args2) {
    auto search = m_registry.find(func);
    if (search != m_registry.end()) {
        auto args_equal = search->second(args1, args2);
        return args_equal;
    }
    throw std::runtime_error("Comparator not found for function");
}

template <typename Arg>
bool compare_arg(void* arg1, void* arg2) {
    // convert a `void*` argument to `Arg`
    auto as_arg = [](void* arg) -> const Arg& {
        return *static_cast<Arg*>(arg);
    };
    return as_arg(args1) == as_arg(args2);
}

// Peel off first element and compare, then call recursively
template <typename... Args> struct Comparator {
    bool operator()(void** args1, void** args2) {
        return (compare_arg<Args>(*(args1++), *(args2++)) && ...);
    }
};
```

Graph Comparison - Comparison

```
class CompareArgsRegistry {
public:
    // Attempt to insert a function into the registry, generating a
    // corresponding comparator based on the function signature.
    template <typename... Args> void register(void(fcn)(Args...)) {
        auto erased_fcn = (void*)fcn;
        auto search = m_registry.find(erased_fcn);
        if (search == m_registry.end()) {
            m_registry[erased_fcn] = Comparator<Args...>();
        } else {
            // we can verify the registered function here, or simply no-op
        }
    }

    // Compares two sets of type-erased arguments to the same function pointer
    // by finding the registered comparator for that function.
    bool compare(void* func, void** args1, void** args2);

    static CompareArgsRegistry& instance();
private:
    using ComparatorFcn = std::function<bool(void**, void**)>;
    using RegistryT = std::map<void*, ComparatorFcn>;
    RegistryT m_registry{};
};
```

```
bool CompareArgsRegistry::compare(void* func, void** args1, void** args2) {
    auto search = m_registry.find(func);
    if (search != m_registry.end()) {
        auto args_equal = search->second(args1, args2);
        return args_equal;
    }
    throw std::runtime_error("Comparator not found for function");
}

template <typename Arg>
bool compare_arg(void* arg1, void* arg2) {
    // convert a `void*` argument to `Arg`
    auto as_arg = [](void* arg) -> const Arg& {
        return *static_cast<Arg*>(arg);
    };
    return as_arg(args1) == as_arg(args2);
}

// Peel off first element and compare, then call recursively
template <typename... Args> struct Comparator {
    bool operator()(void** args1, void** args2) {
        return (compare_arg<Args>(*(args1++), *(args2++)) && ...);
    }
};
```

```
// Finally, we can compare our `cudaKernelNodeParams::kernelParams`
const auto& registry = CompareArgsRegistry::instance();
bool kernel_params_equal = registry.compare(params_a.kernelParams,
                                           params_b.kernelParams);
```

Coroutine

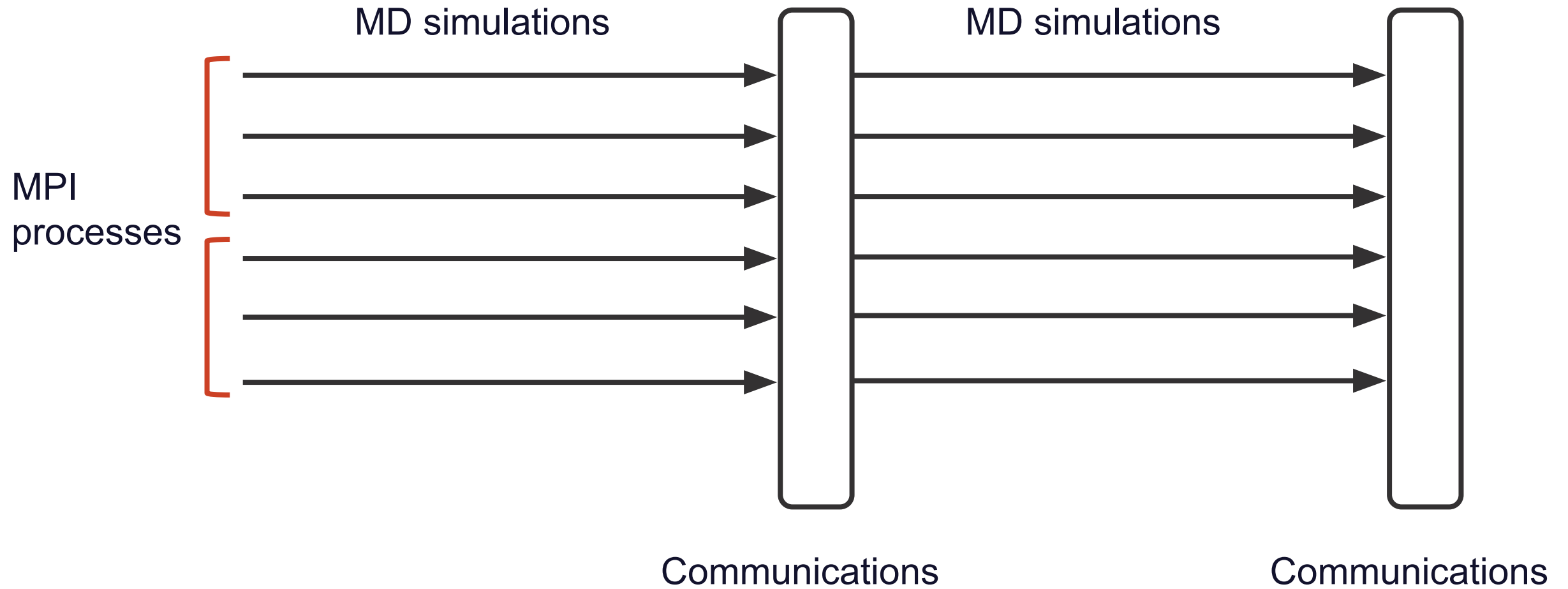
Optimizing for throughput

- Throughput can be increased by scheduling concurrent simulations to the same GPU
 - We can think of total simulations per time per resource
 - Alternatively, we can target high GPU utilization, which is a more convenient metric, as long as we don't perform 'extra' work.
- There are a number of options to achieve this type of concurrency - threads, MPI + CUDA MPS, or kernel interleaving
 - The engineering and maintenance costs of **threads** can be rather high. Compared to MPS, threading also is more challenging for the CUDA driver to handle.
 - **MPI + MPS** (Multi-Process Service) requires more CPUs per GPU and can bottleneck with a high number of small kernels
 - **Kernel interleaving** typically restructuring the program to overlap synchronization points, typically this involves significant and often undesirable code changes
 - As we'll see later, with the use of coroutines we can achieve this with minimal restructuring

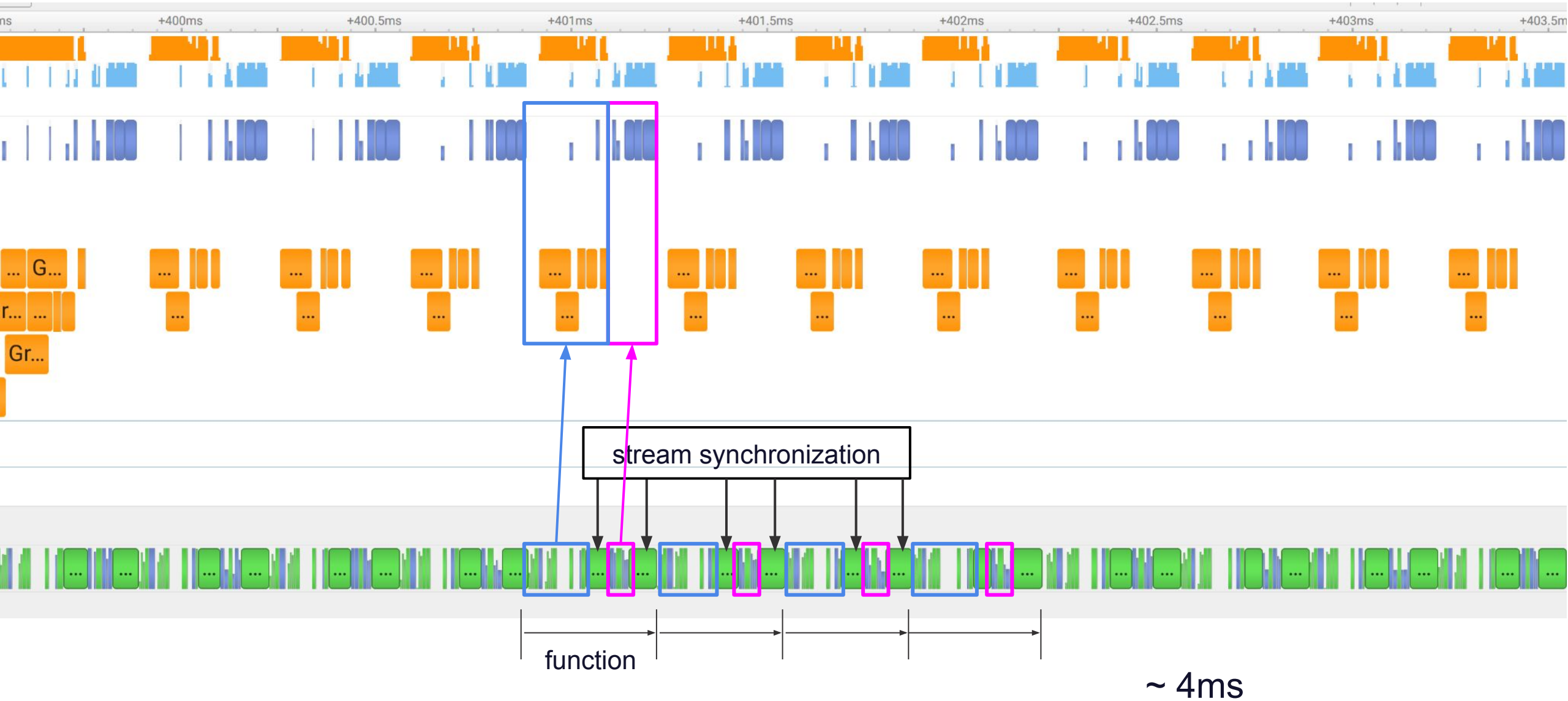
FEP+ has built-in source of concurrency

- Concurrency can be limited by the data itself - we need lots of ideally load-balanced simulations to be requested at the same time
 - Fortunately, FEP+ has built-in concurrency that is nearly perfect for this use
 - Each perturbation contains 12+ MD simulations of almost identical size, so load-balance will not be an issue
 - The most expensive ‘production’ simulations are already time-sliced in the same process, and optionally spread into chunks across a number of MPI processes, due to the need for communication.
 - These can be easily rearranged for concurrency
 - Some smaller ‘equilibration’ simulations are independent and launched separately
 - Requires restructuring, or we can just use MPS
- A combination of MPS + coroutines allows us the flexibility to optimize both the fully independent equilibration simulation and the synchronized production stages

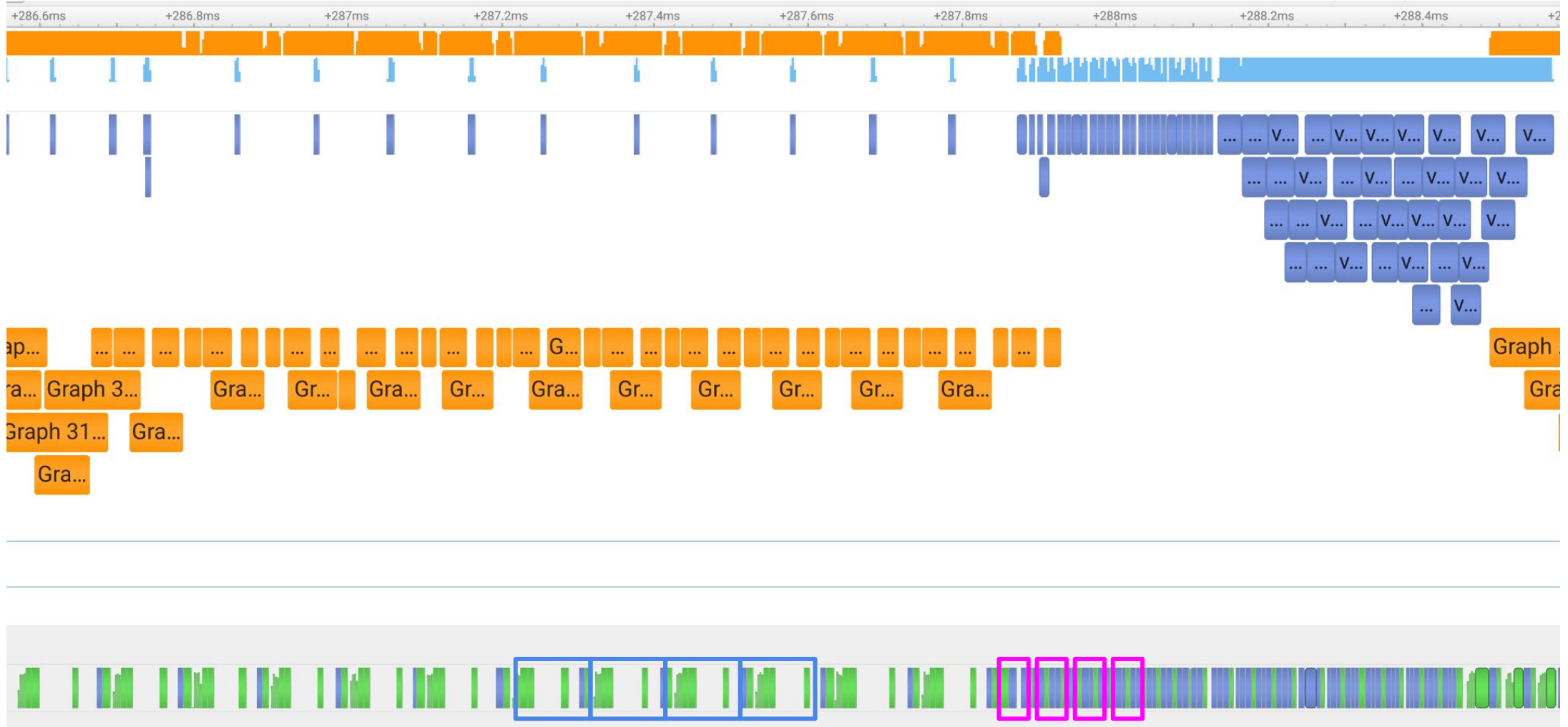
FEP+ has built-in source of concurrency



Sovent leg without coroutine



Solvent leg with coroutine



~2 ms

Coroutines

Coroutines can be described as “Functions that can be paused”

- Those proficient in Python will likely recognize ‘generators’, which use `yield` statements to return control flow to the calling function
 - Control can then be resumed from the yield statement by calling `next` on the generator
 - These generators are a stripped-down example of coroutines
- General-purpose coroutines can be more complex, allowing control flow to be yielded to any other coroutine, and allowing complex communication
 - For our purposes, we need to yield to the top-level caller, and we don’t require additional communication
- We use the boost coroutine library
 - https://www.boost.org/doc/libs/1_84_0/libs/coroutine2/doc/html/index.html
 - This gives us ‘stackful asymmetric coroutines’, allowing us to yield from any point in the program back to the original caller

Coroutine code sample

Coroutines allow us to overlap computation from multiple simulations - simply yield before synchronization points

```
// in top-level function that manages replicas
std::vector<coroutine_t::pull_type> pull_vector;
for (ReplicaPtr& r : replicas) {
    auto coroutine_fcn = [&](coroutine_t::push_type& sink) {
        // Sets/unsets coroutine sink. We could also pass to `apply` below.
        ScopedCoroutineHandle handle(r->sim(), sink);
        // main per-replica function
        r->sim().apply();
    };
    v_pull.emplace_back(coroutine_fcn);
}

auto check_and_enter_coroutine = [](auto& pull) -> bool {
    if (pull) pull(); // enter coroutine if necessary
    return bool(pull); // work still remains
}
for (bool work_remains = true; work_remains;) {
    work_remains = std::accumulate(pull_vector.begin(), pull_vector.end(),
                                   false, check_and_enter_coroutine);
}

// many places inside sim::apply
do_some_cuda_work(cuda_stream);

// before syncing the stream, check if
// this is a coroutine invocation
if (has_sim_sink()) {
    call_sink(); // if so, yield
}
cudaStreamSynchronize(cuda_stream);

...

do_some_more_cuda_work(cuda_stream);

// yield again
if (has_sim_sink()) {
    call_sink(); // if so, yield
}
cudaStreamSynchronize(cuda_stream);
```

Performance

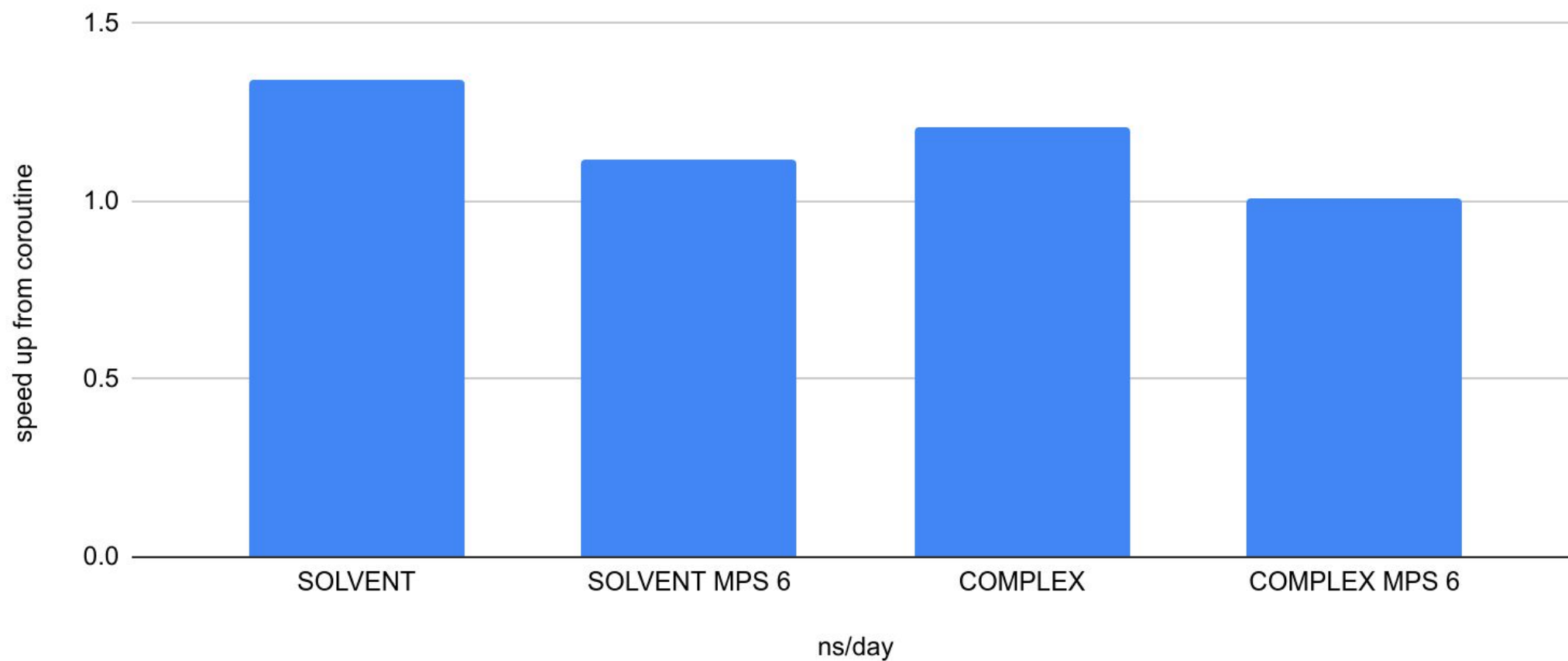
- Benchmarked on an A100-40GB

ns/day	latest without coroutine	latest	speed up from coroutine
SOLVENT	226	304	1.34
SOLVENT MPS 6	805	905	1.12
COMPLEX	110	134	1.21
COMPLEX MPS 6	201	204	1.01

- Solvent leg simulations are very small - no protein
- Many small kernels leads to underutilization, and thus more benefit from coroutine
- A mix of coroutine + MPS is better than MPS alone

Performance

speed up from coroutine vs. ns/day



Mapped memory

Mapped memory

Mapped memory: A page-locked memory that is accessible to the device

- allocated with `cudaHostAlloc/cudaHostRegister/cudaMallocHost`.
- CUDA kernels can write directly to mapped memory - no need to first writing to device memory and then copying to host memory with an additional `cudaMemcpy`, thus also the name *zero copy*.
- Helps in situations where latency matters (host CUDA API latency, PCIe latency, etc).

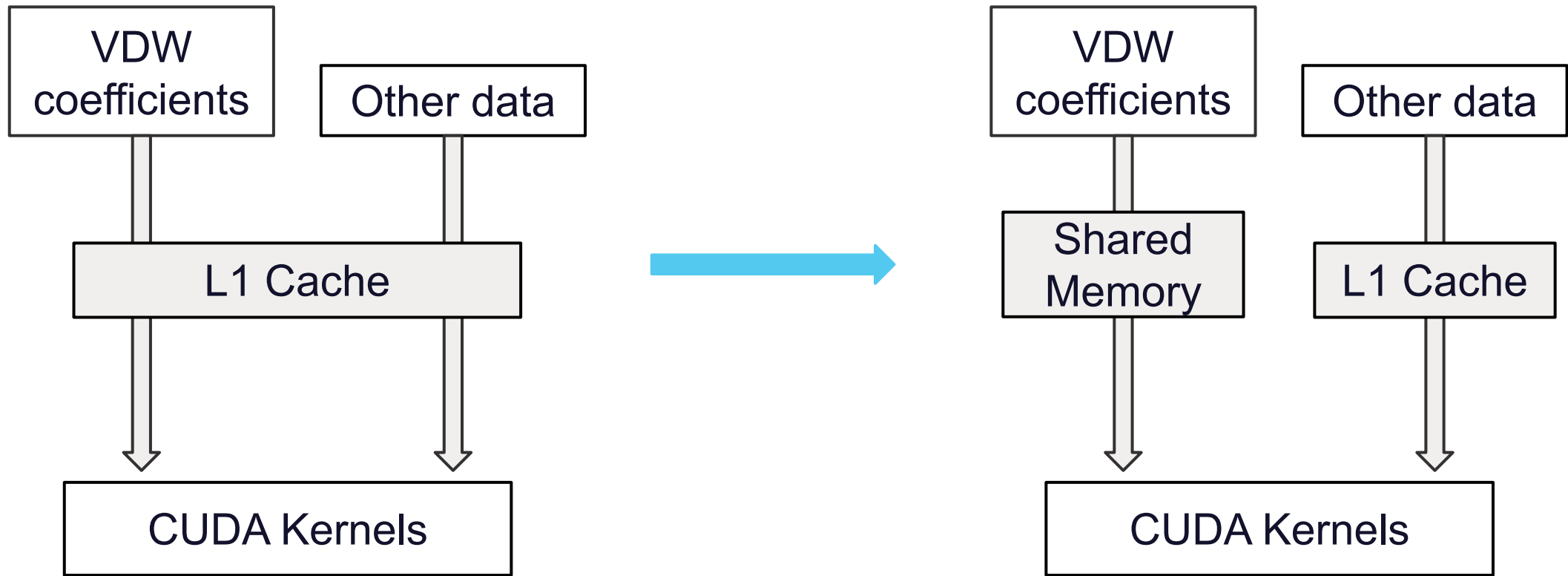
Shared memory

Shared memory

- **L1 cache**
 - High bandwidth, low latency on-chip cache.
- **Shared memory**
 - Programmable L1 cache (that has to be CUDA-programmed).
- The same VDW coefficients are needed for computation repeatedly in the near term kernel in DESMOND.
- Previously they are loaded directly from texture. In principle the L1 cache hit rate for them should already be high. However these loads compete with loads for other types of data, resulting in worse cache thrashing.
- Now at the beginning of the kernel all the VDW coefficients are loaded from texture once and are then stored in shared memory. Later in the kernel they are loaded from shared memory directly.

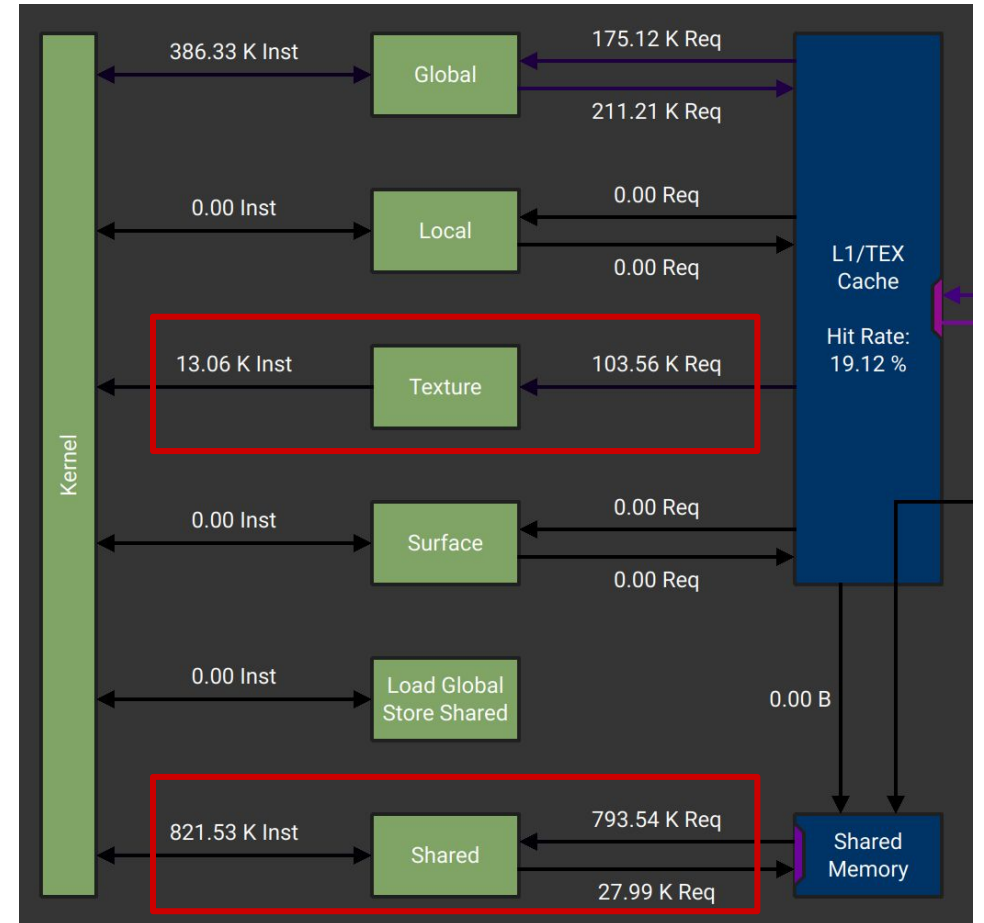
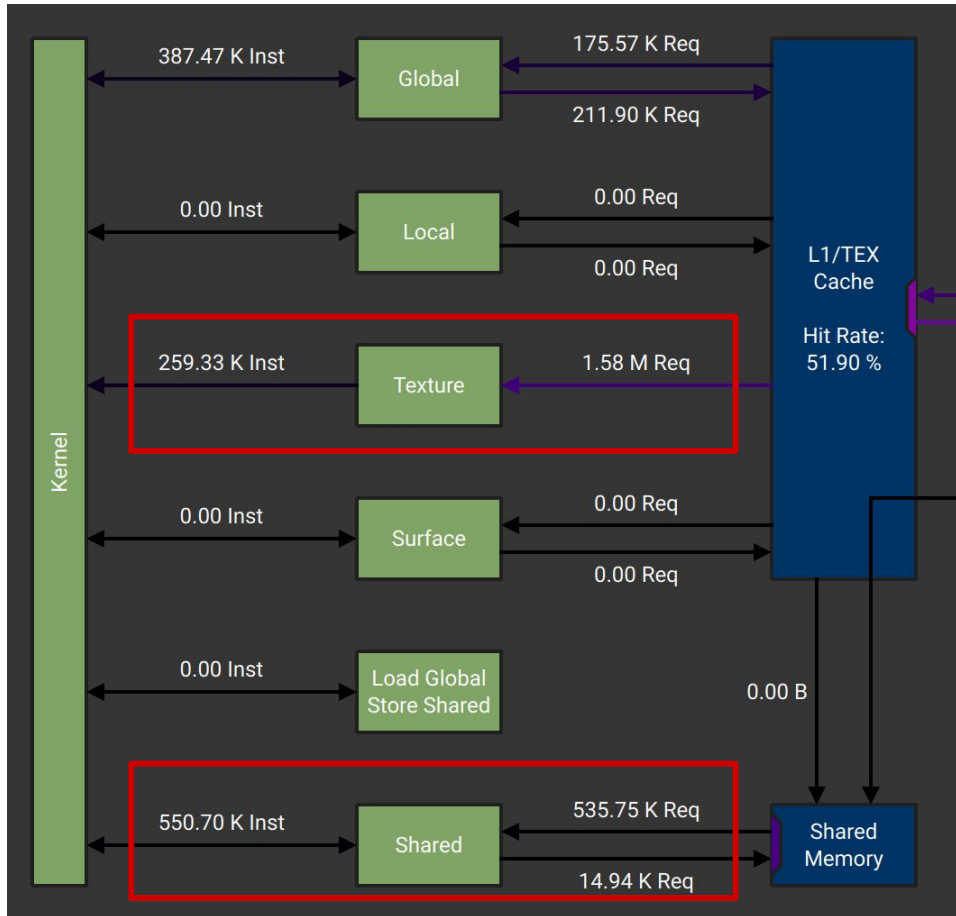
Shared memory

- Reduces L1 cache thrashing by leaving the L1 cache to the other loads: they have lower L1 hit rate and have larger footprint that they cannot fit in shared memory.



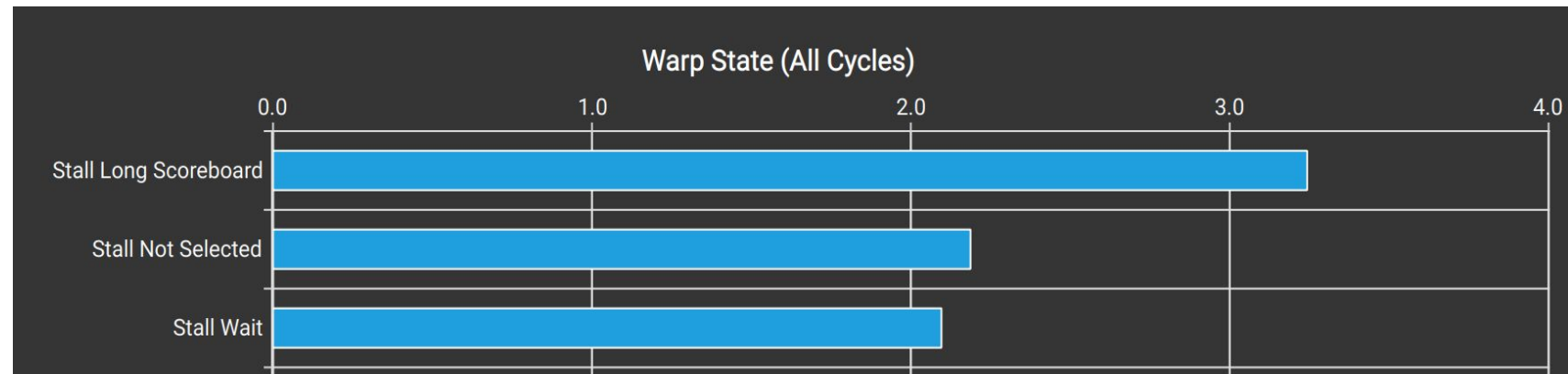
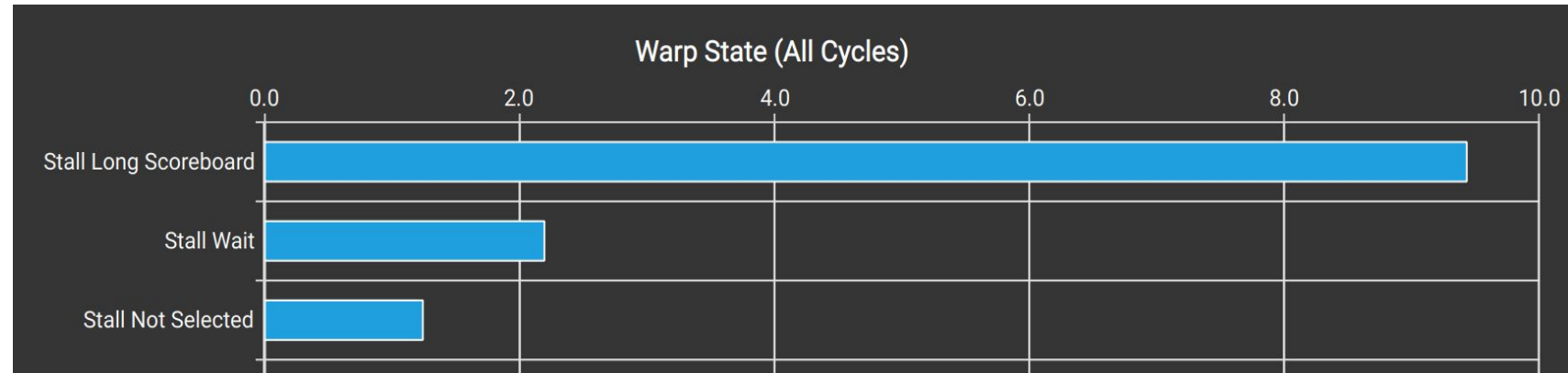
Shared memory

- As expected, we have got more shared memory loads but less loads from texture.



Shared memory

- Stall long scoreboard is greatly reduced.
- Total kernel cycle: 109,398 -> 86,664 (21% faster).
- Stall long scoreboard:
9.43->3.34
- Stall short scoreboard:
0.70->0.87



Applying more (new) CUDA features

More CUDA features

- Conditional nodes in CUDA graph
 - Less number of CUDA graphs to manage, since one graph can be used for multiple workflow needs. Less memory needed.
 - Less overall engineering cost.
- CUDA dynamic parallelism (CDP)
 - Launching kernels, allocating CUDA memory, etc, from within a CUDA kernel.
 - Results in (even) less number of host-device synchronizations, since the most of the things can happen on the device.
 - This then results in (even) better overall performance.
- ... and potentially more

Thanks for Attending

Please feel free to contact us

- Ellery Russell: ellery.russell@schrodinger.com
- Jiqun Tu: jtu@nvidia.com

For more information, please see:

- [Technical Blog: “Constructing CUDA Graphs with Dynamic Parameters”](#)