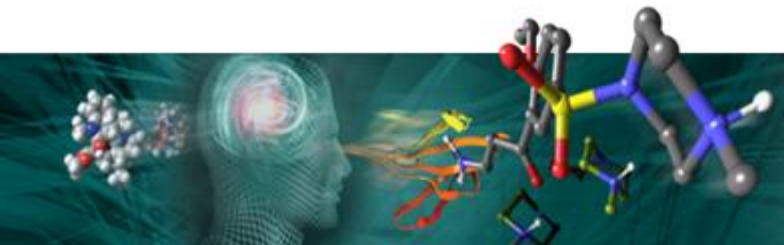
Software Architecture to Facilitate CUDA Development

Peter S. Shenkin, NVidia GTC, May, 2012

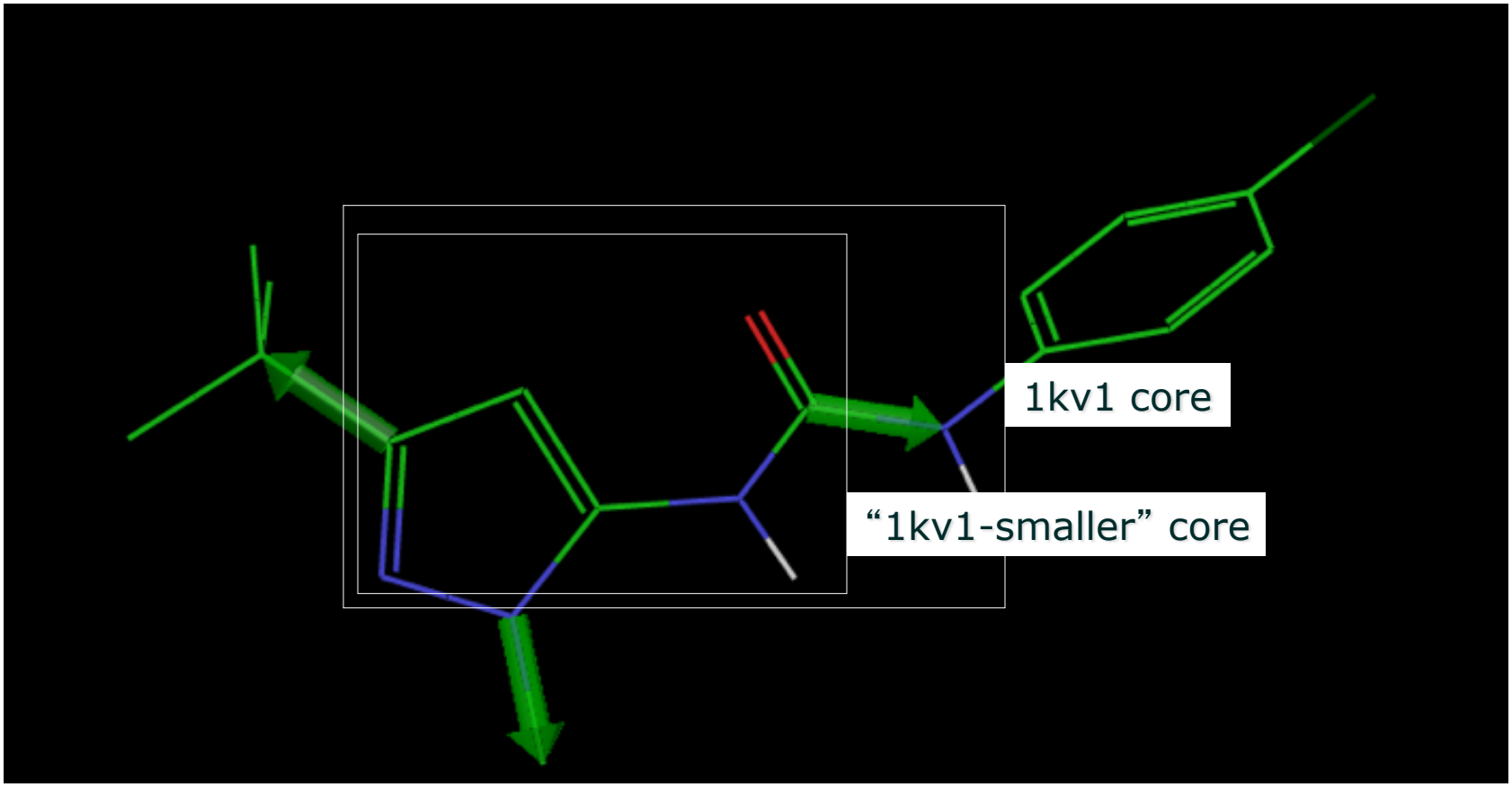


SCHRÖDINGER.

The Core-Hopping Application

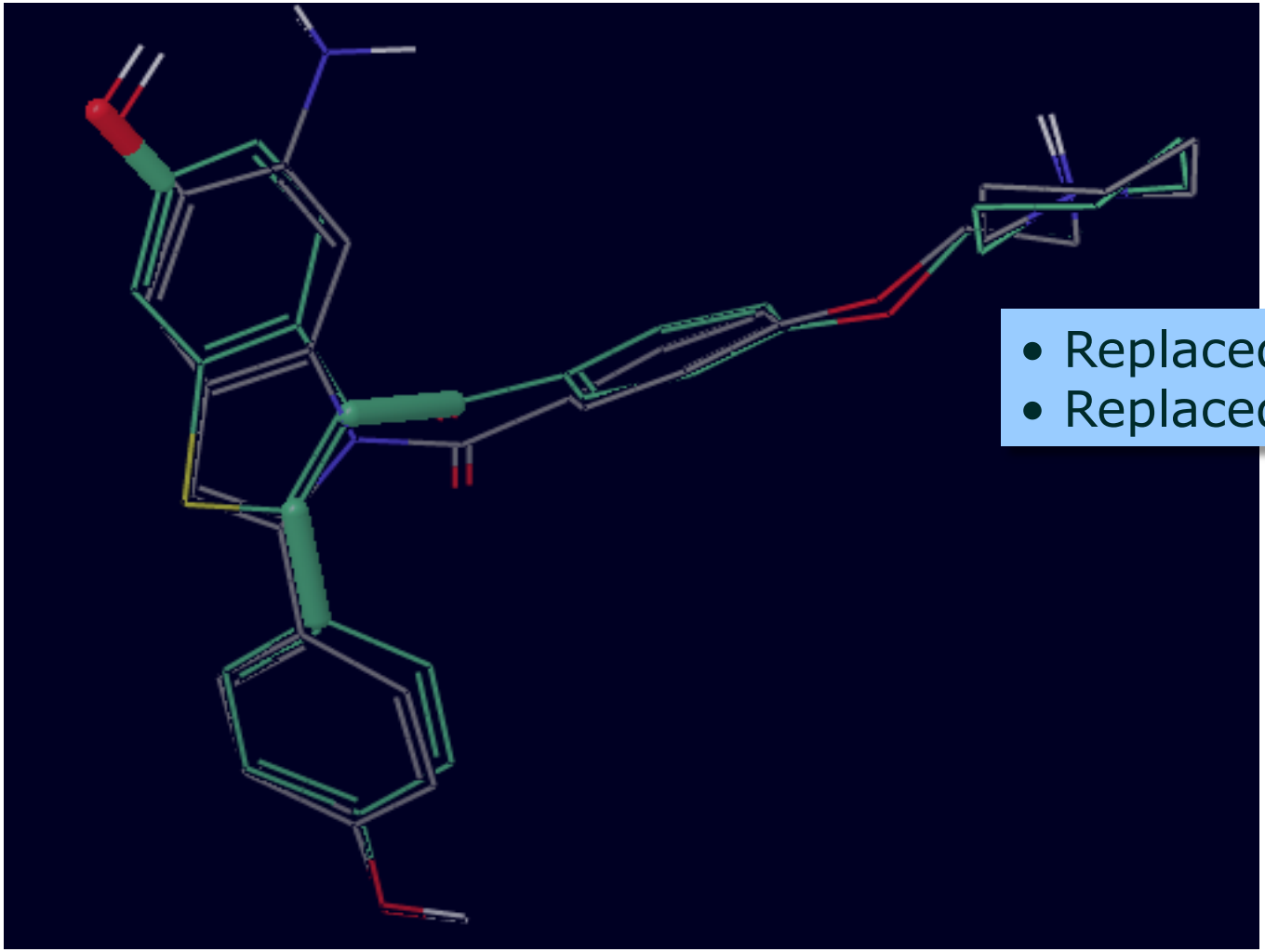
- Application area: drug design
- What it does
  - Find a replacement for the central portion of a molecule
  - … keeping the peripheral parts (R groups) in place
  - … while making “chemical sense”
  - We may add “linkers” between a small core and an R group
- Designed as a fast interactive desktop application
- Why would one do such a thing?
  - Increase drug efficacy
  - Improve “ADMET” properties
    - (Absorption, Distribution, Metabolism, Excretion, Toxicity)
  - Find new IP

Define Core in a “Template” Molecule



- Two ways shown, to emphasize user choice

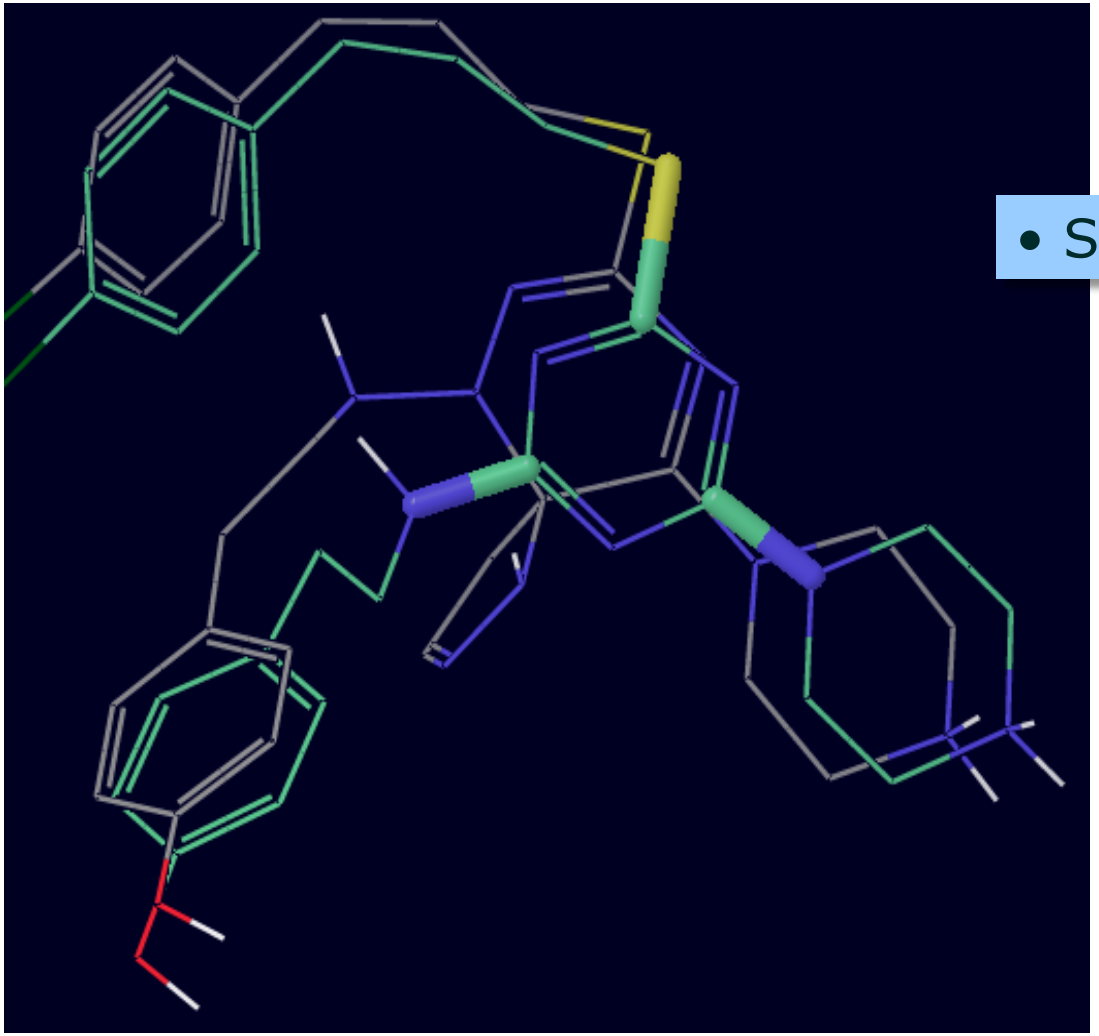
Result: 1err: olap= 0.95 rel gscore= -1.37



- Replaced C with N
- Replaced S with C

SCHRÖDINGER.

Result: 1erb: olap= 0.80, rel gscore= -0.96



- Spiro core!

SCHRÖDINGER.

The Architecture

- The architecture
  - Workflow engine independent of application code
  - Workflow engine is coprocessor-agnostic
    - All the CUDA logic is in the CUDA worker threads
  - Multithreaded using Qthreads; C++
  - Application stages are essentially plug-ins
- Advantages:
  - CPU and CUDA stages in separate functions
    - Facilitates benchmarking, testing, debugging
    - Can use completely different CPU and CUDA algorithms
  - Mix and match CUDA vs. CPU stage by stage, based on:
    - Perception of environment at run-time
    - User-specified run-time options
- Disadvantages
  - Constrains application architecture to a pipeline
  - Good for new development, probably not legacy code

The Architecture (Example)

- AppTaskFlow: the pipeline
- Tasks: The stages in the pipeline; there are queues in between
- ThreadTypes: The allowed compute environments
- RunTimeTaskFlow: Contiguous tasks w. same ThreadTypes
  - A single thread runs a single RunTimeTaskFlow at a time

	AppTaskFlow					
	RunTim…	RunTim…	RunTimeTaskFlow		RunTim…	RunTim…
Tasks: / Thread Types:	Input, Pre-process	Add Linkers	Sample Space	Match Template	(…)	Post-process, Output
Single-threaded	X	X	X	X	X	X
Multi-threaded		X	X	X	X	
CUDA			X	X		

SCHRÖDINGER.

The Architecture (Example)

```
// Define a workflow:
AppTaskFlow core_hopping = AppTaskFlow( "core_hopping",
 q_min, q_max ) // min, max ∑queue sizes between stages

// Add tasks to the workflow; worker fns. unspecified:
Task *pre = core_hopping.addTask( "preprocess" )
Task *link = core_hopping.addTask( "add_linkers" )
Task *space = core_hopping.addTask( "sample_space )
. . .

// Define thread types; semantics unspecified:
ThreadType single_cpu = ThreadType( "single_cpu" )
ThreadType multi_cpu = ThreadType( "multi_cpu" )
ThreadType cuda = ThreadType( "cuda" )
```

The Architecture (Example)

```
// Add user-defined worker functions to tasks. These
//  are callbacks, and are where the work takes place:
pre->addWorker( single, pre_worker_cpu )

link->addWorker( single, link_worker_cpu )
link->addWorker( multi, link_worker_cpu )

space->addWorker( single, space_worker_cpu )
space->addWorker( multi, space_worker_cpu )
space->addWorker( cuda, space_worker_cuda )
. . .

// At run-time, tell each thread type how many
//  simultaneous threads it is allowed:
single.setMaxThreads( 1 )
multi.setMaxThreads( nproc ) // Or cmdline option
cuda.setMaxThreads( 2 * nGPU ) // Could be 0!
```

The Architecture (Worker Functions)

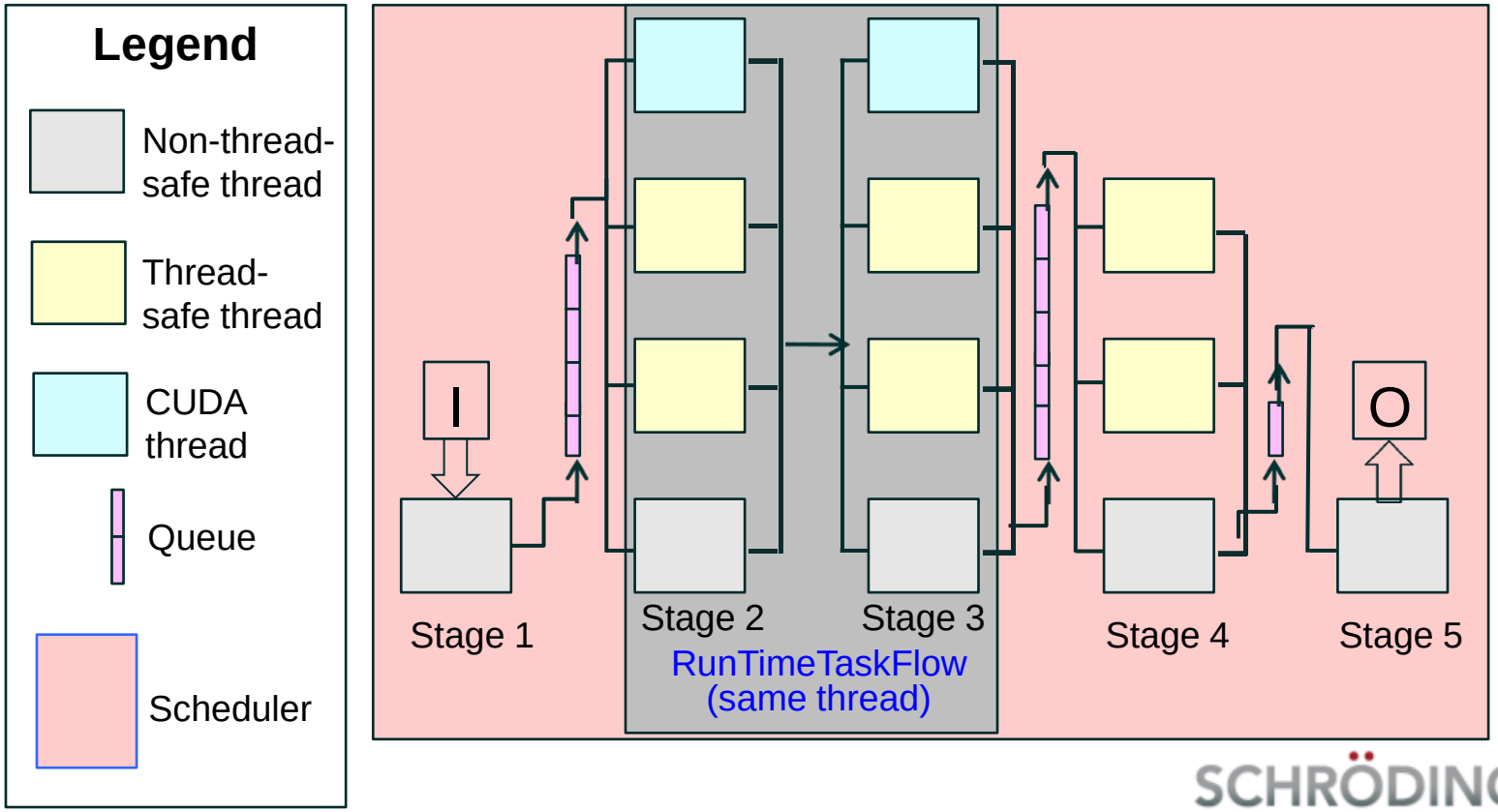
- The above is all the workflow code the user has to write
- Of course, the user has to write the worker functions
- CUDA code is in functions like **space_worker_cuda**

```
// Worker functions all share the same prototype.
//  They pass data to and from the driver (to be
//  enqueued) in structs that are (void*) from the
//  outside:
typedef enum{ worker_ok, worker_finished, worker_died }
 Worker_result;
typedef void *Worker_data;
typedef Worker_result (*Worker)( Worker_data input,
 Worker_data *output )
```

SCHRÖDINGER.

At Run Time:

- The driver runs as many simultaneous threads as it can
- It stacks up “work parcels” between successive stages in fifos
- Dispatch schedule for new threads considers:
  - Available compute environments and waiting stages
  - Queue sizes… (etc.)



SCHRÖDINGER.

Benchmarks

Test system (home-built):

- i7/930, 2.7 GHz processor
  - 4 physical cores, hyperthreaded (looks like 8)
  - Hyperthreading has benchmarking implications
- 12 Gb RAM
- 8-lane PCIe motherboard
- SSD drive
- GPUs:
  - 2x Tesla 2075
  - N.B. In previous experiments, GeForce does as well

Results

GPUs	Threads	kCores/hr
2	9	959
1	9	674
0	9	322

Results

GPUs	Threads	kCores/hr	CPU %
2	9	959	50%
1	9	674	63%
0	9	322	98%

Results

GPUs	Threads	kCores/hr	CPU %
2	9	959	50%
1	9	674	63%
0	9	322	98%
0	8	321	94%
0	7	293	84%
0	6	271	75%
0	5	246	63%
0	4	222	50%
0	3	169	38%
0	2	115	25%

Results

GPUs	Threads	kCores/hr	CPU %	Speedup
2	9	959	50%	4.3
1	9	674	63%	2.7
0	9	322	98%	
0	8	321	94%	
0	7	293	84%	
0	6	271	75%	
0	5	246	63%	
0	4	222	50%	
0	3	169	38%	
0	2	115	25%	

Results

GPUs	Threads	kCores/hr	CPU %	Speedup
2	9	959	50%	4.3
1	9	674	63%	2.7
0	5	246	63%	
0	4	222	50%	

- Both GPGPU runs are pre/postprocessor limited
  - These are single-threaded, and run in the same thread
  - The thread “ping-pongs” between pre and post stages
- Elapsed time in this thread is same as total elapsed time
- Single-CPU run is slower but still pre/post limited
  - Hyperthreading slows the single threads
- Algorithm is quite fast even on CPU
  - This is a new, optimized CPU implementation
  - Compared to our legacy codes, this is *much* faster

Summary

- We have produced an architecture that facilitates CUDA development for a class of applications
- We have developed an application based on this architecture
- The application exhibits good CUDA performance

Acknowledgments

- Pat Lorton, application development
  - Including CUDA “heavy lifting”
- *Thanks to:*
  - Joe Landman, Scalable Informatics
  - Jason Chen, Exxact Corp.
  - … for access to their GPGPU deskside products
  - … which gave the same performance as internal benchmarks