GPU TECHNOLOGY CONFERENCE

# High-Productivity CUDA Programming

Cliff Woolley, Sr. Developer Technology Engineer, NVIDIA

# HIGH-PRODUCTIVITY PROGRAMMING

# High-Productivity Programming

**What does this mean? What's the goal?**

- **Do Less Work (Expend Less Effort)**
- **Get Information Faster**
- **Make Fewer Mistakes**

# High-Productivity Programming

- Do Less Work (Expend Less Effort)
  - Use a specialized programming language
  - Reuse existing code
- Get Information Faster
- Make Fewer Mistakes

# High-Productivity Programming

- Do Less Work (Expend Less Effort)
- Get Information Faster
  - Debugging and profiling tools
  - Rapid links to documentation
  - Code outlining
  - Type introspection/reflection
- Make Fewer Mistakes

# High-Productivity Programming

- Do Less Work (Expend Less Effort)
- Get Information Faster
- Make Fewer Mistakes
  - Syntax highlighting
  - Code completion
  - Type checking
  - Correctness checking

# High-Productivity Programming

**What kinds of tools exist to meet those goals?**

- Specialized programming languages, smart compilers
- Libraries of common routines
- Integrated development environments (IDEs)
- Profiling, correctness-checking, and debugging tools

# HIGH-PRODUCTIVITY CUDA PROGRAMMING

# High-Productivity CUDA Programming

## What's the goal?

- Port existing code quickly
- Develop new code quickly
- Debug and tune code quickly
- …Leveraging as many tools as possible

**This is really the same thing as before!**

# High-Productivity CUDA Programming

## What kinds of tools exist to meet those goals?

- Specialized programming languages, smart compilers
- Libraries of common routines
- Integrated development environments (IDEs)
- Profiling, correctness-checking, and debugging tools

# HIGH-PRODUCTIVITY CUDA:

## Programming Languages, Compilers

# GPU Programming Languages

| | |
|---|---|
| Numerical analytics | MATLAB, Mathematica, LabVIEW |
| Fortran | OpenACC, CUDA Fortran |
| C | OpenACC, CUDA C |
| C++ | CUDA C++, Thrust, Hemi, ArrayFire |
| Python | Anaconda Accelerate, PyCUDA, Copperhead |
| .NET | CUDAfy.NET, Alea.cuBase |

developer.nvidia.com/language-solutions

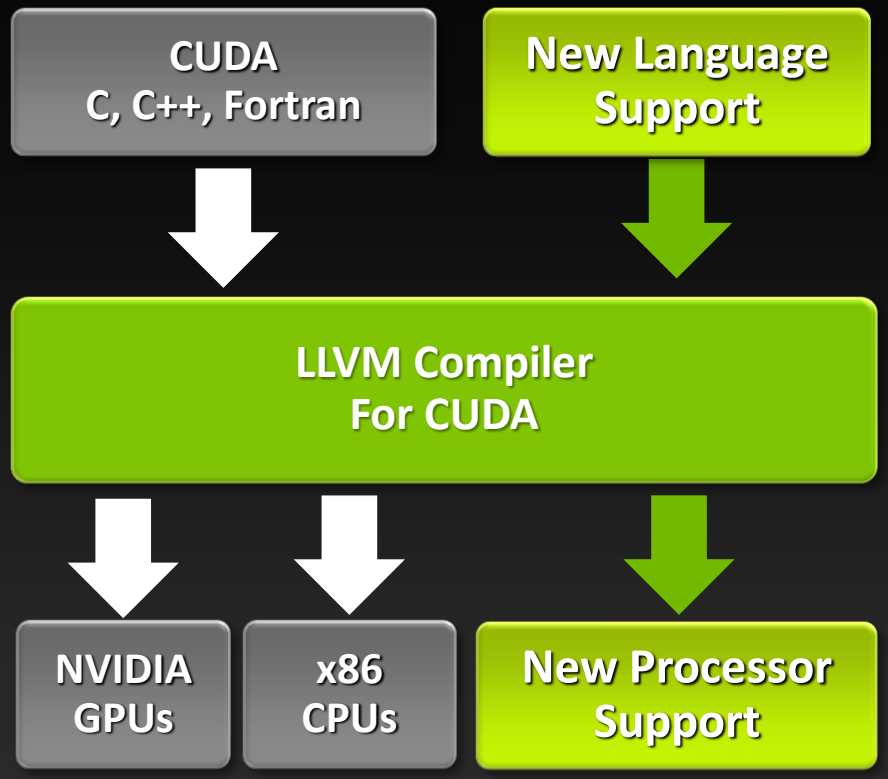
# Opening the CUDA Platform with LLVM


CUDA compiler source contributed
to open source LLVM compiler project

SDK includes specification documentation,
examples, and verifier

**Anyone can add CUDA support
to new languages and processors**

Learn more at
developer.nvidia.com/cuda-llvm-compiler



# HIGH-PRODUCTIVITY CUDA:

## GPU-Accelerated Libraries

# GPU Accelerated Libraries

## "Drop-in" Acceleration for your Applications

NVIDIA cuBLAS

NVIDIA cuSPARSE

NVIDIA NPP

NVIDIA cuFFT

MAGMA

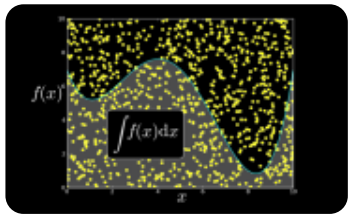
Matrix Algebra on GPU and Multicore

CULA | tools

GPU Accelerated Linear Algebra

GPU VSIPL

Vector Signal Image Processing



NVIDIA cuRAND

ROGUE WAVE SOFTWARE

IMSL Library

CenterSpace NMath

ArrayFire

Building-block Algorithms

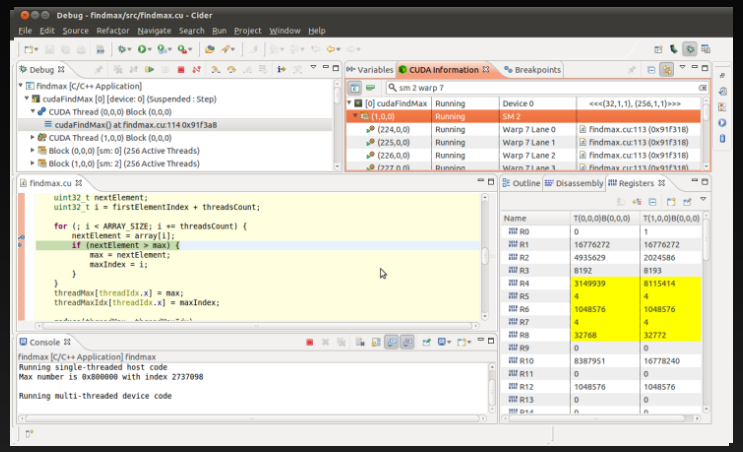
Thrust

C++ Templated Parallel Algorithms

# HIGH-PRODUCTIVITY CUDA:

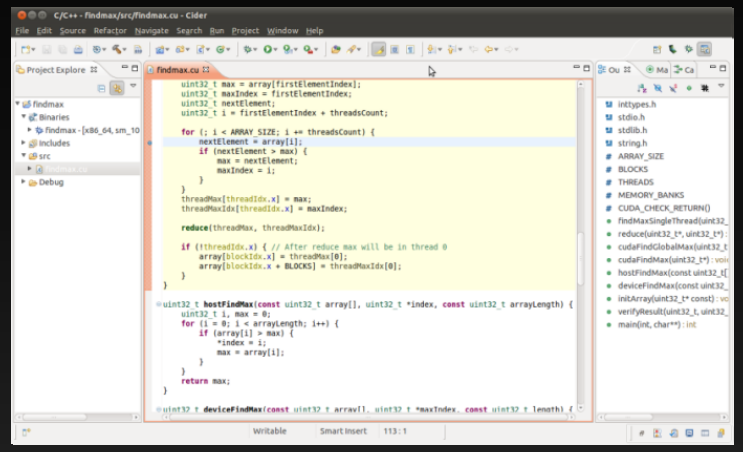
## Integrated Development Environments

# NVIDIA® Nsight,™ Eclipse Edition
## for Linux and MacOS



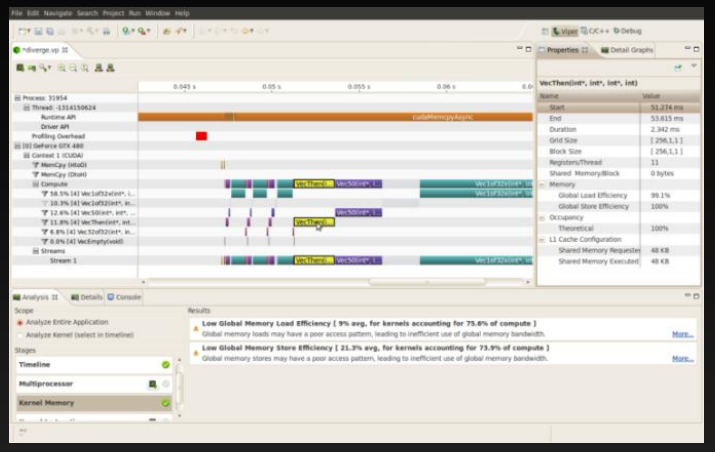
### CUDA-Aware Editor

- Automated CPU to GPU code refactoring
- Semantic highlighting of CUDA code
- Integrated code samples & docs



### Nsight Debugger

- Simultaneously debug CPU and GPU
- Inspect variables across CUDA threads
- Use breakpoints & single-step debugging

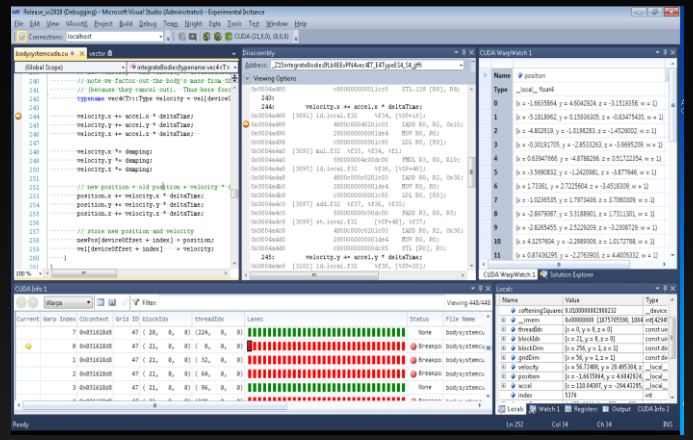


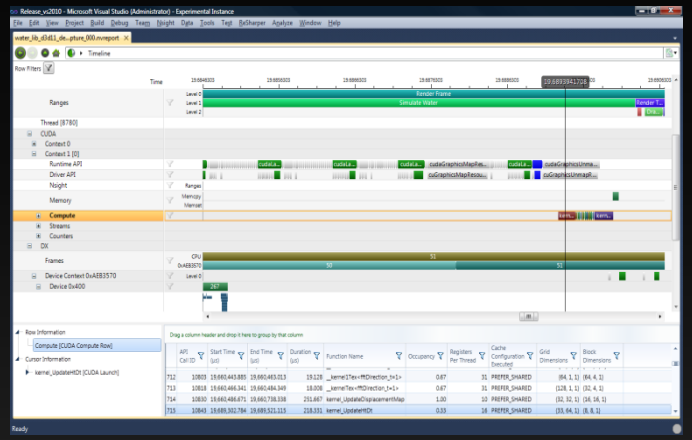
### Nsight Profiler

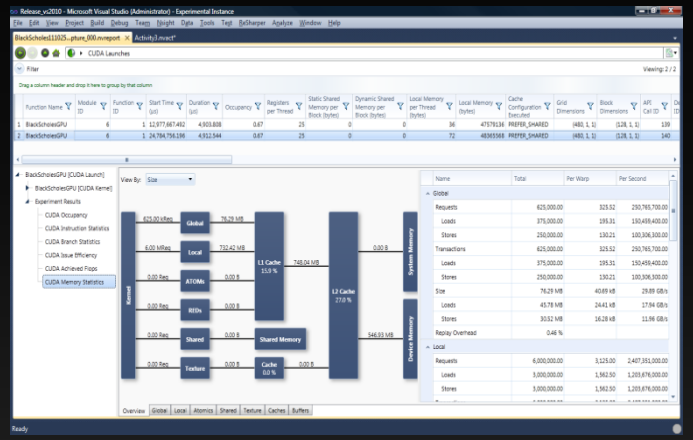
- Quickly identifies performance issues
- Integrated expert system
- Source line correlation

developer.nvidia.com/nsight

# NVIDIA® Nsight,™ Visual Studio Edition







## CUDA Debugger

- Debug CUDA kernels directly on GPU hardware
- Examine thousands of threads executing in parallel
- Use on-target conditional breakpoints to locate errors

## CUDA Memory Checker

- Enables precise error detection

## System Trace

- Review CUDA activities across CPU and GPU
- Perform deep kernel analysis to detect factors limiting maximum performance

## CUDA Profiler

- Advanced experiments to measure memory utilization, instruction throughput and stalls

# HIGH-PRODUCTIVITY CUDA:

## Profiling and Debugging Tools

# Debugging Solutions

## Command Line to Cluster-Wide

**NVIDIA Nsight**
Eclipse & Visual Studio Editions

**NVIDIA CUDA-GDB**
for Linux & Mac

**NVIDIA CUDA-MEMCHECK**
for Linux & Mac

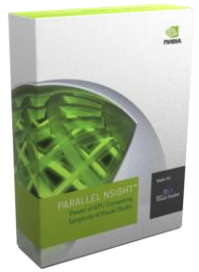
**Allinea DDT with CUDA**
Distributed Debugging Tool

**TotalView for CUDA**
for Linux Clusters

developer.nvidia.com/debugging-solutions

# Performance Analysis Tools

## Single Node to Hybrid Cluster Solutions



**NVIDIA Nsight**
Eclipse & Visual Studio Editions


**NVIDIA Visual Profiler**

Vampir Trace Collector

Tuning and Analysis Utilities

TAU Performance System

PAPI

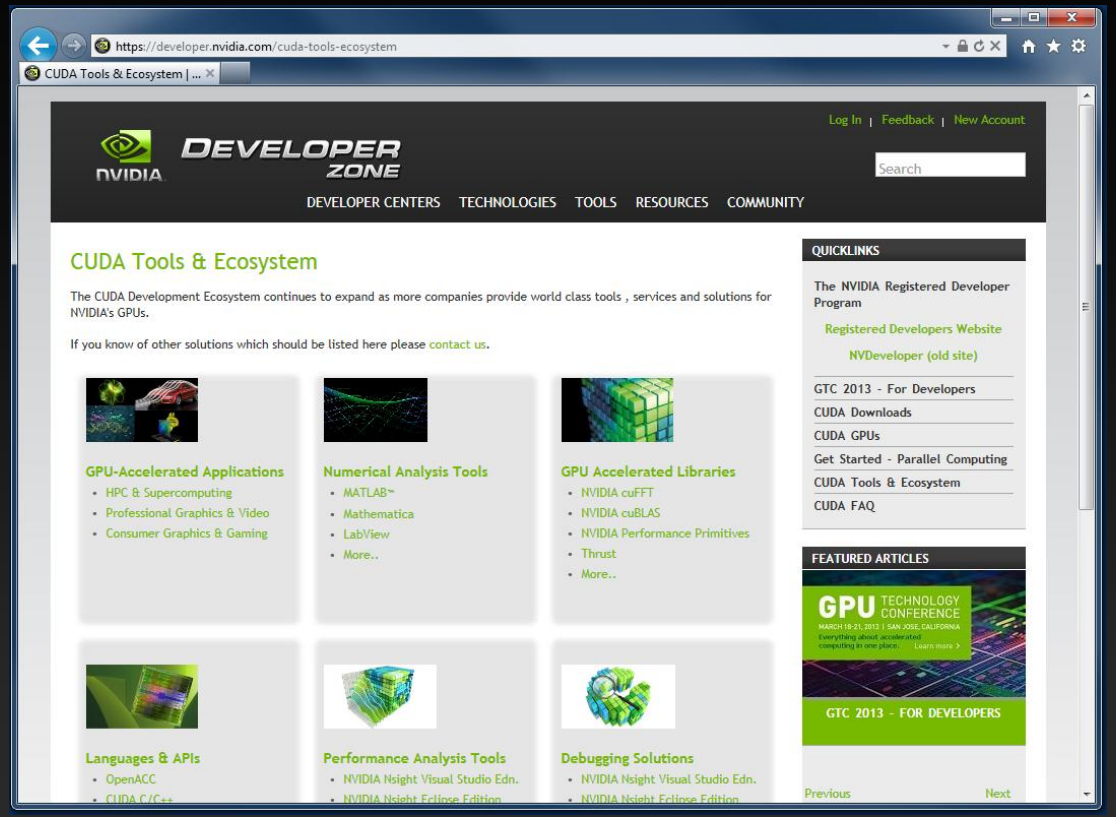
PAPI CUDA Component

HPCToolkit

Under Development

developer.nvidia.com/performance-analysis-tools

# Want to know more? Visit DeveloperZone



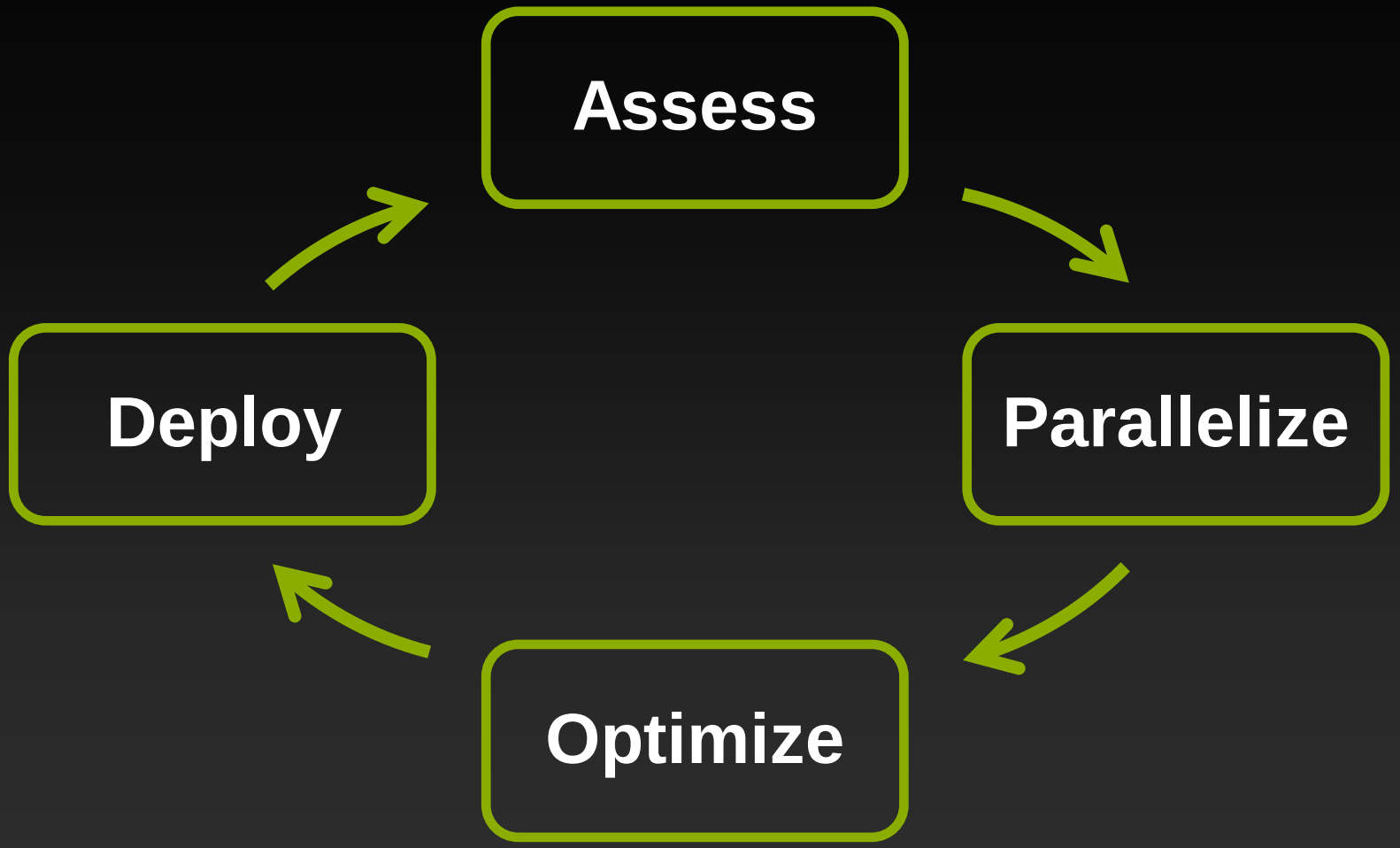

**developer.nvidia.com/cuda-tools-ecosystem**

# High-Productivity CUDA Programming

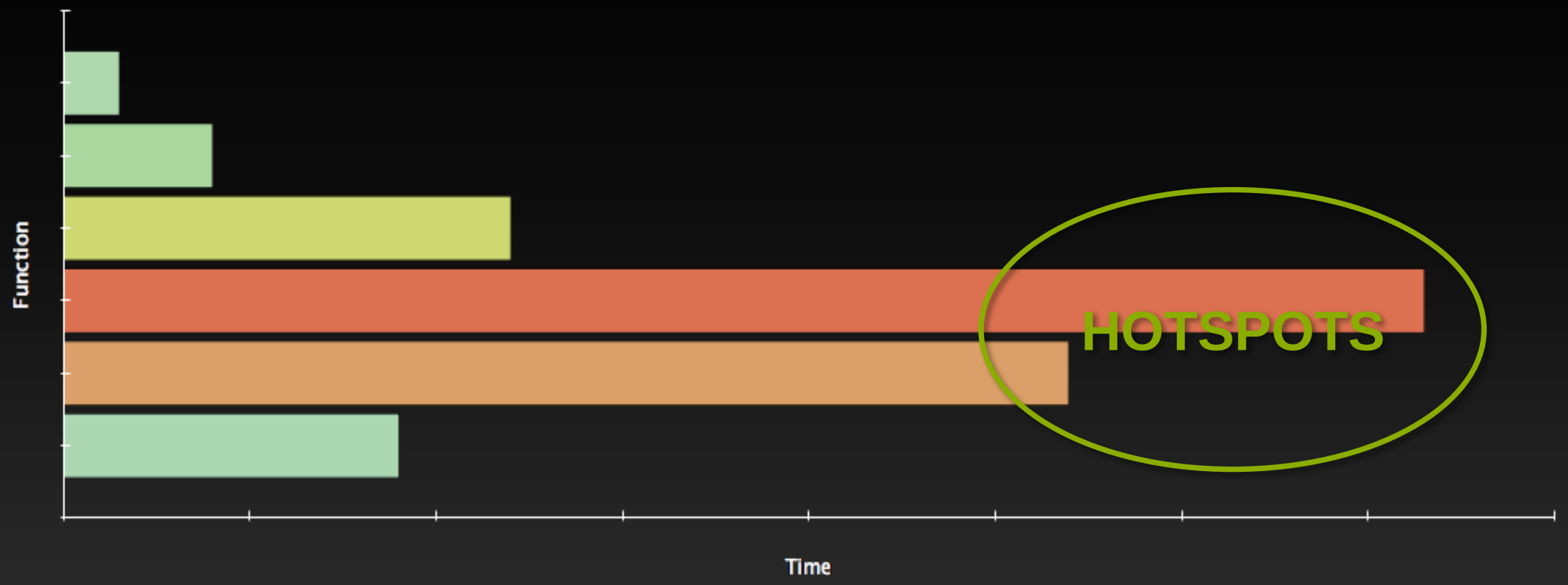
- Know the tools at our disposal
- Use them wisely
- Develop *systematically*

# SYSTEMATIC CUDA DEVELOPMENT

# APOD: A Systematic Path to Performance



# Assess




- Profile the code, find the hotspot(s)
- Focus your attention where it will give the most benefit

# Parallelize

Applications

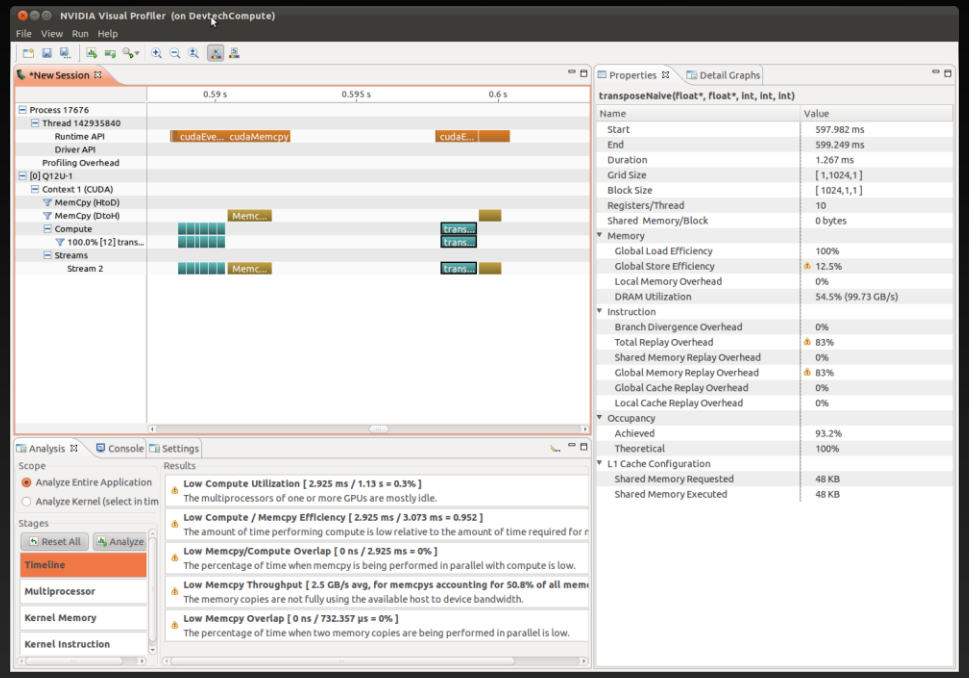
Libraries

OpenACC Directives

Programming Languages

# Optimize

- Profile-driven optimization
- Tools:
  - nsight NVIDIA Nsight IDE
  - nvvp NVIDIA Visual Profiler
  - nvprof Command-line profiling



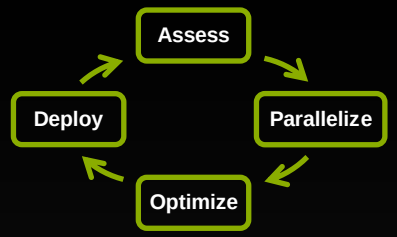
# Deploy

## Productize

- Check API return values
- Run cuda-memcheck tools

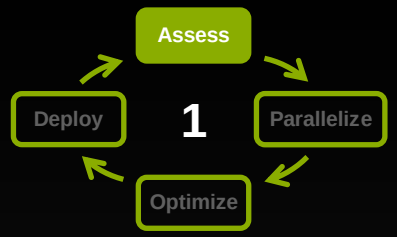
- Library distribution
- Cluster management

Early gains
Subsequent changes are evolutionary




# SYSTEMATIC CUDA DEVELOPMENT

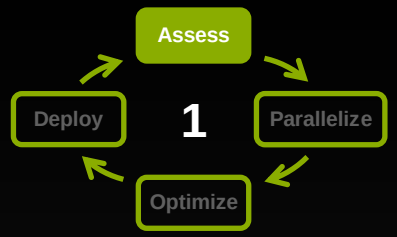
## APOD Case Study




# APOD CASE STUDY

## Round 1: Assess

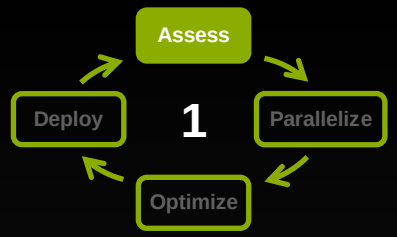
# Assess




| | Function Name | Module ID | Function ID | Count | Device % | Device Time (µs) | Min (µs) | Avg (µs) | Max (µs) | Context ID |
|---|---|---|---|---|---|---|---|---|---|---|
| 1 | spmv_kernel_v0<int=256> | 8 | 3 | 59 | 35.96 | 457,088.169 | 7,700.182 | 7,747.257 | 7,809.367 | 1 |
| 2 | jacobi_smooth_kernel_v0<int=256> | 6 | 4 | 29 | 1.49 | 18,899.315 | 650.599 | 651.701 | 653.224 | 1 |
| 3 | dot_kernel_v0<int=256> | 5 | 2 | 58 | 0.36 | 4,553.489 | 53.601 | 78.508 | 89.953 | 1 |
| 4 | axpbypcz_kernel<int=256> | 5 | 5 | 28 | 0.35 | 4,442.227 | 157.857 | 158.651 | 159.394 | 1 |
| 5 | l2_norm_kernel_v0<int=256> | 7 | 1 | 30 | 0.30 | 3,793.612 | 124.801 | 126.454 | 128.098 | 1 |
| 6 | axpby_kernel<int=256> | 5 | 4 | 31 | 0.29 | 3,741.159 | 119.809 | 120.683 | 123.170 | 1 |
| 7 | jacobi_invert_diag_kernel_v0<int=256> | 6 | 1 | 1 | 0.06 | 783.209 | 783.209 | 783.209 | 783.209 | 1 |
| 8 | reduce_kernel<int=256> | 5 | 1 | 58 | 0.03 | 366.597 | 5.984 | 6.321 | 7.168 | 1 |
| 9 | reduce_l2_norm_kernel<int=256> | 7 | 2 | 30 | 0.02 | 291.490 | 9.504 | 9.716 | 10.560 | 1 |

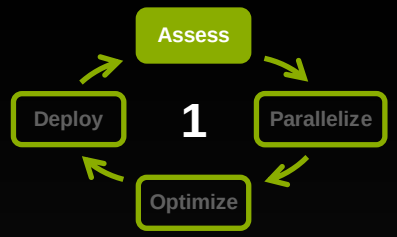
- **Profile the code, find the hotspot(s)**
- **Focus your attention where it will give the most benefit**

# Assess




- We've found a hotspot to work on!
  - What percent of our <u>total time</u> does this represent?
  - How much can we improve it? What is the "speed of light"?

# Assess




| | Function Name | Module ID | Function ID | Count | Device % | Device Time (µs) | Min (µs) | Avg (µs) | Max (µs) | Context ID |
|---|---|---|---|---|---|---|---|---|---|---|
| 1 | spmv_kernel_v0<int=256> | 8 | 3 | [illegible] | [illegible] | [illegible] | [illegible] | 7,747.257 | 7,809.367 | 1 |
| 2 | jacobi_smooth_kernel_v0<int=256> | 6 | [illegible] | 29 | 1.49 | [illegible] | [illegible] | [illegible] | 653.224 | 1 |
| 3 | dot_kernel_v0<int=256> | 5 | 2 | 58 | [illegible] | 4,553.489 | [illegible] | [illegible] | [illegible] | 1 |
| 4 | axpbypcz_kernel<int=256> | 5 | 5 | 28 | 0.35 | [illegible] | [illegible] | [illegible] | 159.394 | 1 |
| 5 | l2_norm_kernel_v0<int=256> | 7 | 1 | 30 | 0.30 | [illegible] | [illegible] | [illegible] | 128.098 | 1 |
| 6 | axpby_kernel<int=256> | 5 | 4 | 31 | 0.29 | [illegible] | [illegible] | 18,899.315 | [illegible] | 1 |
| 7 | jacobi_invert_diag_kernel_v0<int=256> | 6 | 1 | 1 | 0.06 | 783.209 | [illegible] | 4,553.489 | 783.209 | 1 |
| 8 | reduce_kernel<int=256> | 5 | 1 | 58 | 0.03 | 366.597 | [illegible] | 4,442.227 | 7.168 | 1 |
| 9 | reduce_l2_norm_kernel<int=256> | 7 | 2 | 30 | 0.02 | 291.490 | [illegible] | 3,793.612 | 10.560 | 1 |

~36%?

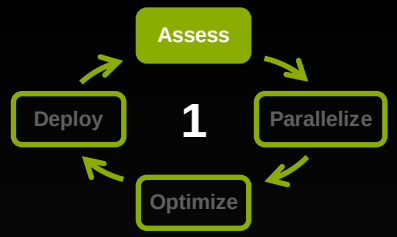
~93%

Device Time (µs)

18,899.315
4,553.489
4,442.227
3,793.612
3,741.159
783.209
366.597
291.490

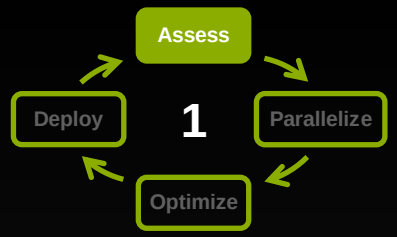
- **What percent of total time does our hotspot represent?**

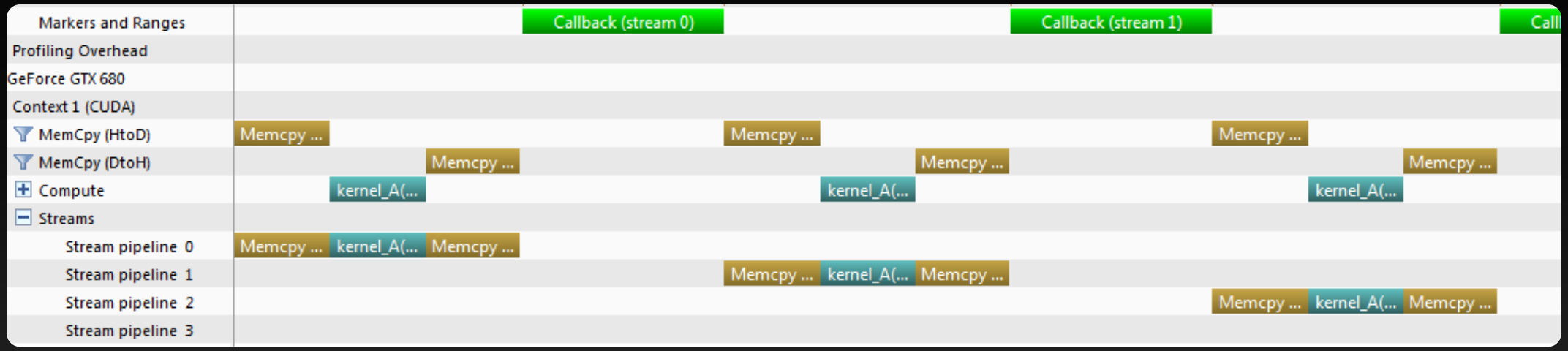
# Assess




- We've found a hotspot to work on!
  - It's 93% of GPU compute time, but only 36% of *total* time
- What's going on during that other ~60% of time?
  - Maybe it's one-time startup overhead not worth looking at
  - Or maybe it's significant – we should check

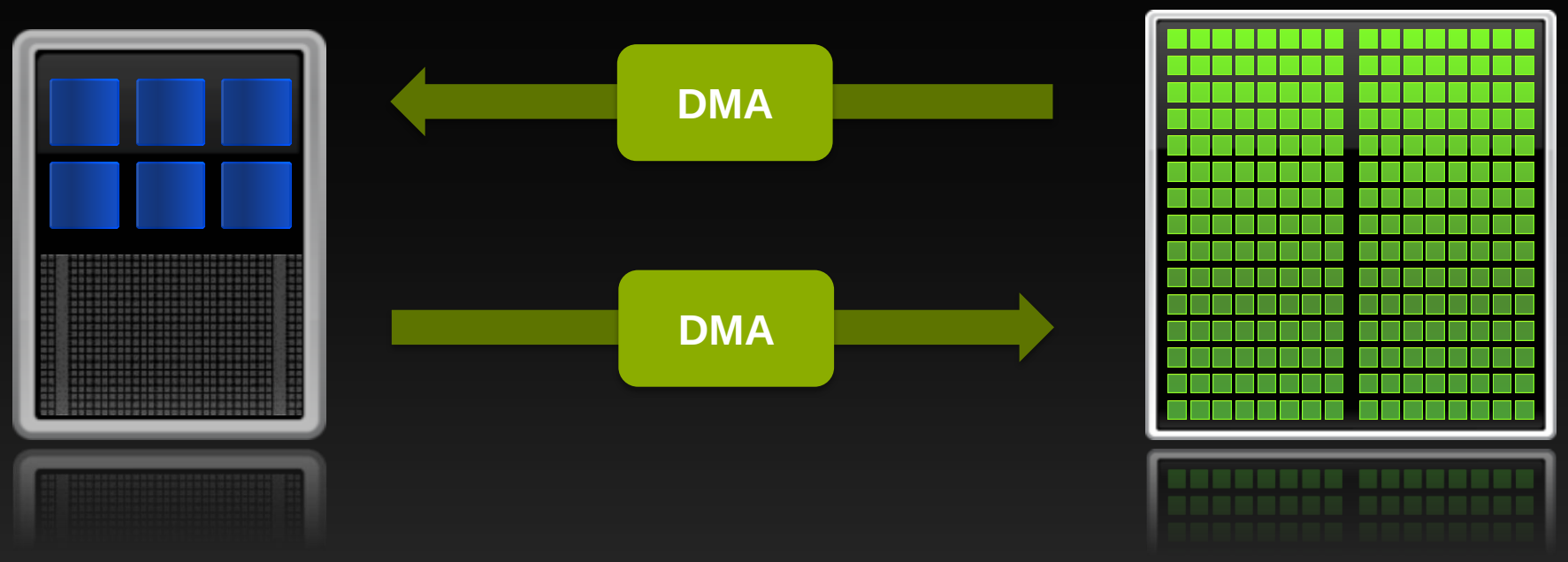
# Assess: Asynchronicity



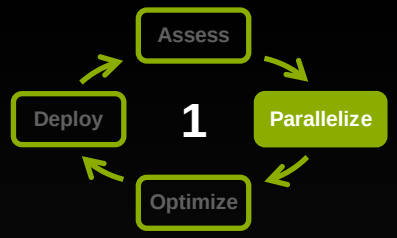




- This is the kind of case we would be concerned about
  - Found the top kernel, but the GPU is mostly idle – *that* is our bottleneck
  - Need to overlap CPU/GPU computation and PCIe transfers

# Asynchronicity = Overlap = Parallelism



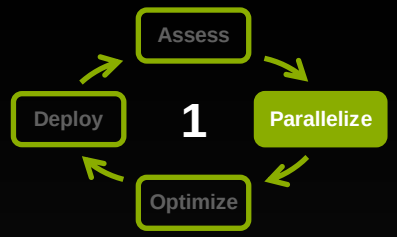

- Heterogeneous system: overlap work and data movement
- Kepler/CUDA 5: Hyper-Q and CPU Callbacks make this fairly easy

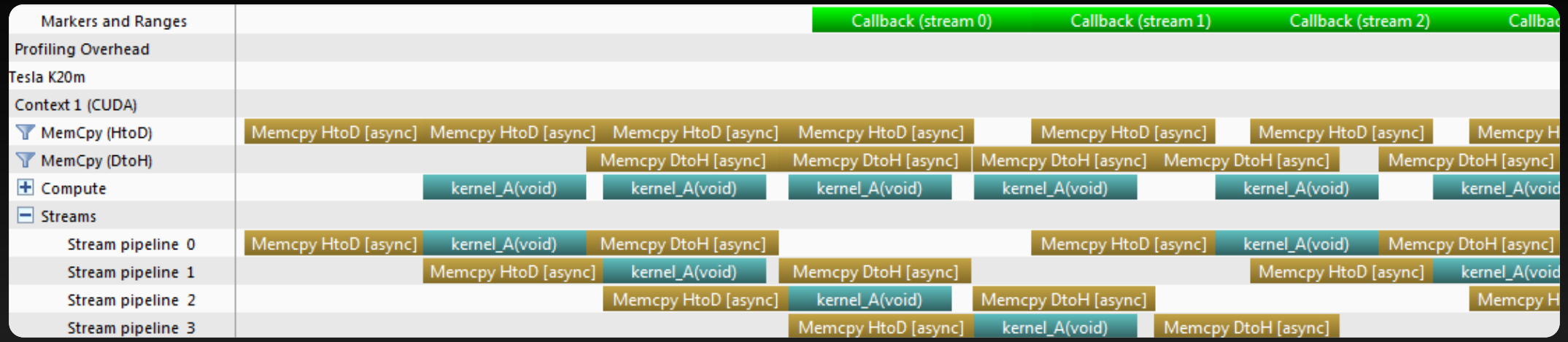



# APOD CASE STUDY

## Round 1: Parallelize

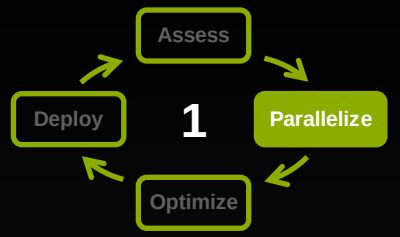
# Parallelize: Achieve Asynchronicity

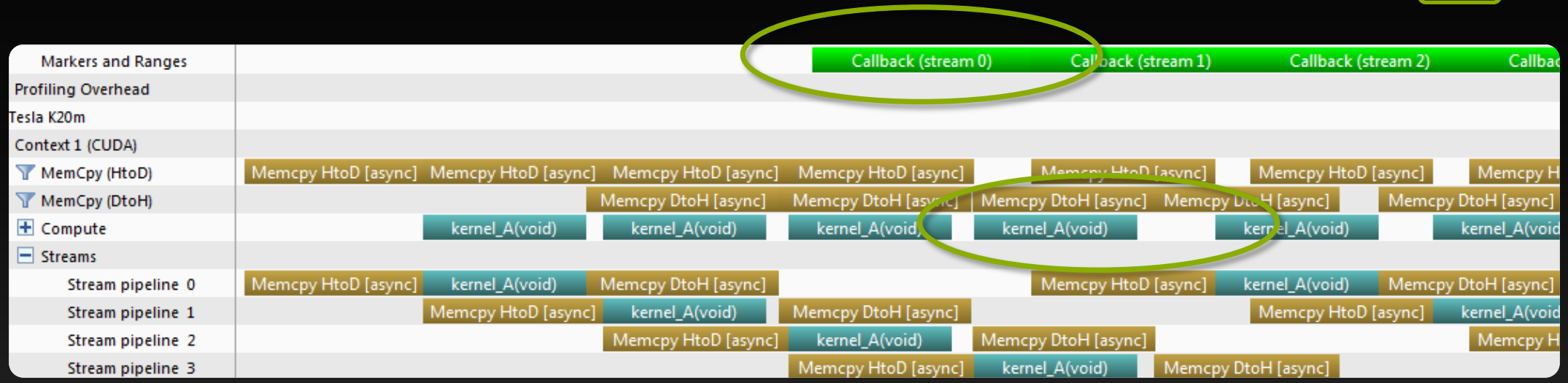



- What we *want* to see is maximum overlap of all engines
- How to achieve it?
  - Use the CUDA APIs for asynchronous copies, stream callbacks
  - Or use CUDA Proxy and multiple tasks/node to approximate this

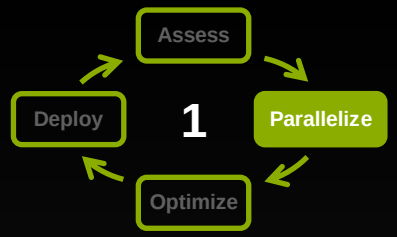
# Parallelize Further or Move On?




- **Even after we fixed overlap, we still have some pipeline bubbles**
  - **CPU time per iteration is the limiting factor here**
  - **So our next step should be to parallelize more**

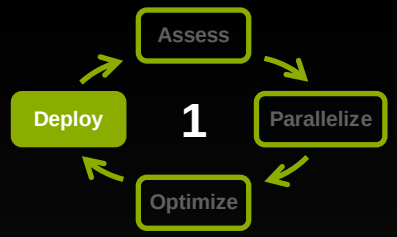
# Parallelize Further or Move On?




## Here's what we know so far:

- We found the (by far) top kernel
- But we also found that GPU was idle most of the time
- We fixed this by making CPU/GPU/memcpy work asynchronous
- We'll need to parallelize more (CPU work is the new bottleneck)
- …And that's before we even think about optimizing the top kernel

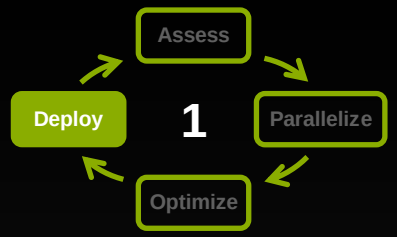
- But we've already sped up the app by a significant margin!
- Skip ahead to Deploy.




# APOD CASE STUDY

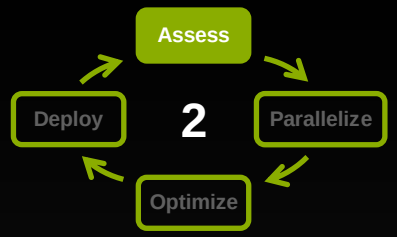
## Round 1: Deploy

# Deploy




- We've already sped up our app by a large margin
- Functionality remains the same as before
  - We're just keeping as many units busy at once as we can
- Let's reap the benefits of this sooner rather than later!

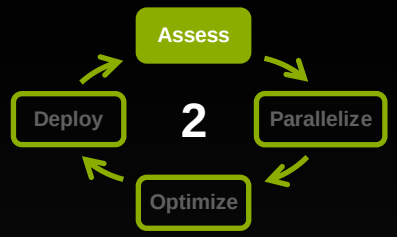
- Subsequent changes will continue to be evolutionary rather than revolutionary

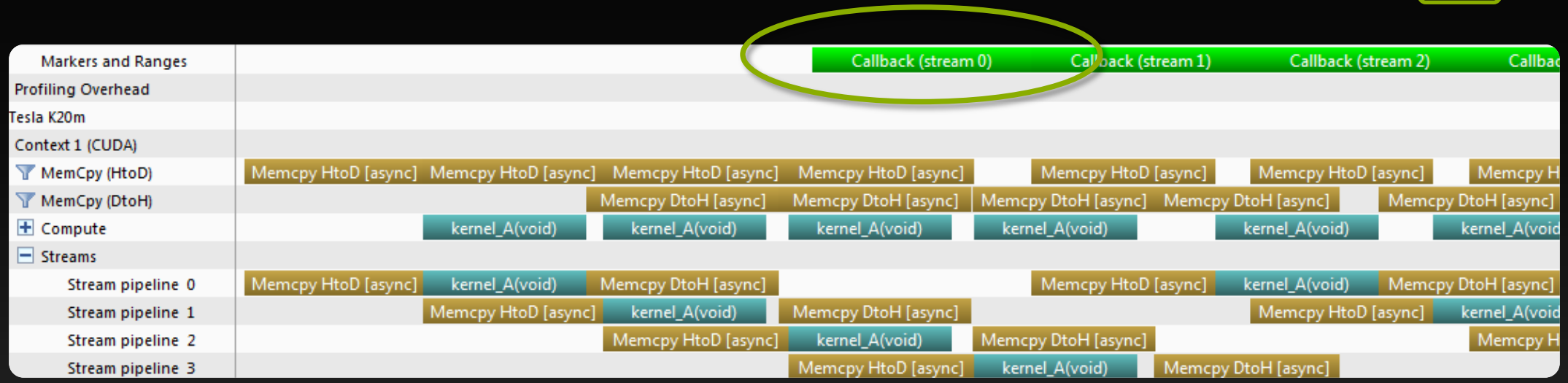



# APOD CASE STUDY

## Round 2: Assess

# Assess

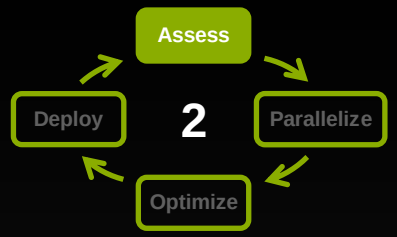



**Our first round already gave us a glimpse at what's next:**

- **CPU compute time is now dominating**
- **No matter how well we tune our kernels or reduce our PCIe traffic at this point, it won't reduce total time even a little**

# Assess

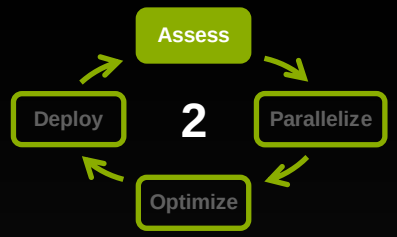



## We need to tune our CPU code somehow

- Maybe that part was never the bottleneck before and just needs to be cleaned up
- Or maybe it could benefit by further improving asynchronicity (use idle CPU cores, if any; or if it's MPI traffic, focus on that)
- We could attempt to vectorize this code if it's not already
- This may come down to offloading more work to the GPU
  - If so, which approach will work best?

**Notice that these basically say "parallelize"**

# Assess: Offloading Work to GPU



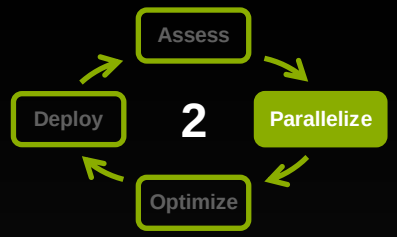

Applications

Libraries

OpenACC Directives

Programming Languages

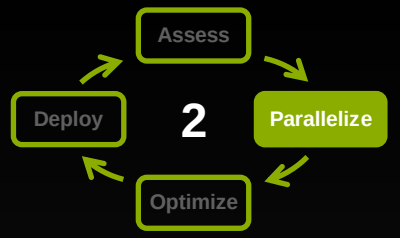
**Pick the best tool for the job**

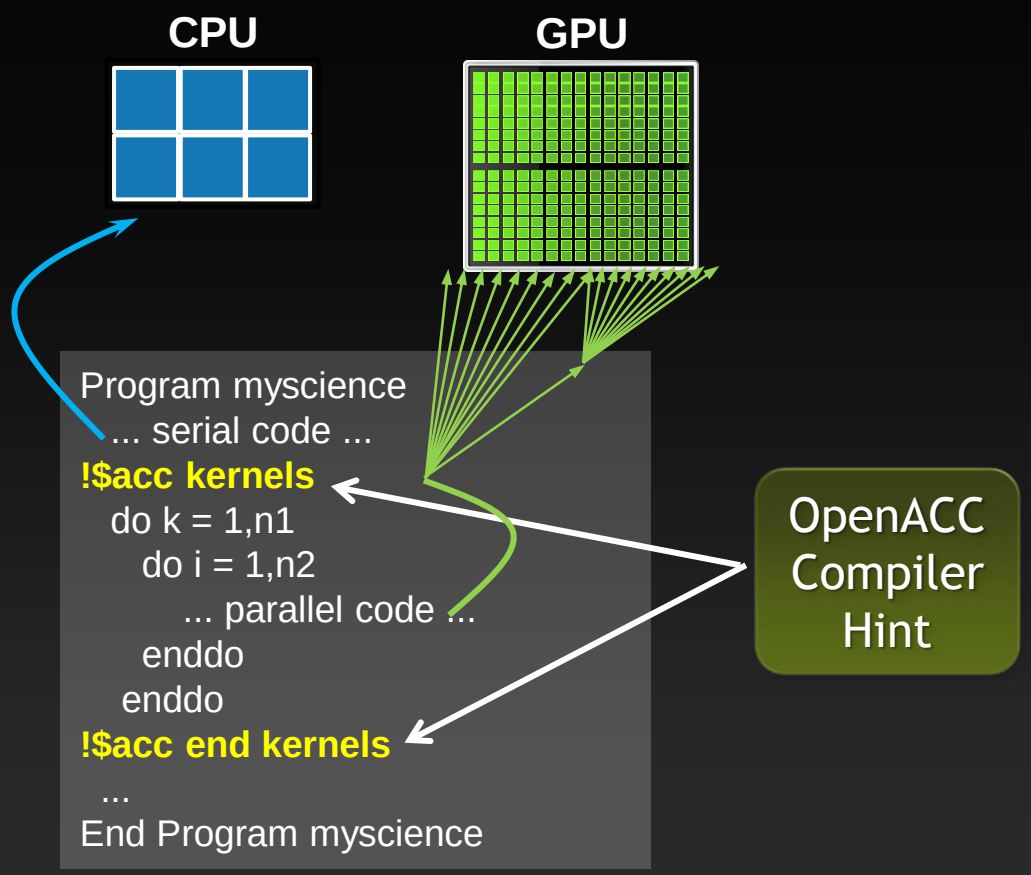



# APOD CASE STUDY

## Round 2: Parallelize

# Parallelize: e.g., with OpenACC




Simple Compiler hints

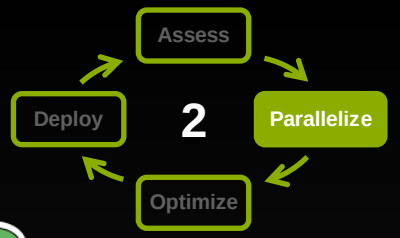
Compiler Parallelizes code

Works on many-core GPUs & multicore CPUs

OpenACC®
DIRECTIVES FOR ACCELERATORS

www.nvidia.com/gpudirectives

# Parallelize: e.g., with Thrust

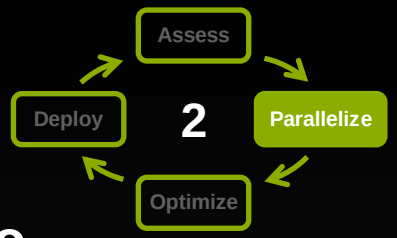



- Similar to C++ STL
- High-level interface
  - Enhances developer productivity
  - Enables performance portability between GPUs and multicore CPUs
- Flexible
  - Backends for CUDA, OpenMP, TBB
  - Extensible and customizable
  - Integrates with existing software
- Open source

```
// generate 32M random numbers on host
thrust::host_vector<int> h_vec(32 << 20);
thrust::generate(h_vec.begin(),
                 h_vec.end(),
                 rand);

// transfer data to device (GPU)
thrust::device_vector<int> d_vec = h_vec;

// sort data on device
thrust::sort(d_vec.begin(), d_vec.end());

// transfer data back to host
thrust::copy(d_vec.begin(),
             d_vec.end(),
             h_vec.begin());
```

thrust.github.com or developer.nvidia.com/thrust

# Parallelize: e.g., with CUDA C



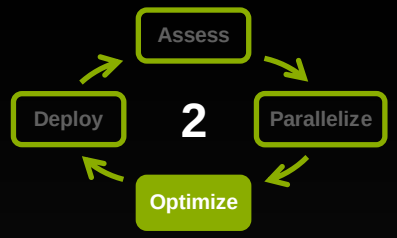

## Standard C Code

```
void saxpy_serial(int n,
                  float a,
                  float *x,
                  float *y)
{

  for (int i = 0; i < n; ++i)
    y[i] = a*x[i] + y[i];
}

// Perform SAXPY on 1M elements
saxpy_serial(4096*256, 2.0, x, y);
```

## Parallel C Code

```
__global__
void saxpy_parallel(int n,
                    float a,
                    float *x,
                    float *y)
{
  int i = blockIdx.x * blockDim.x +
          threadIdx.x;
  if (i < n) y[i] = a*x[i] + y[i];
}

// Perform SAXPY on 1M elements
saxpy_parallel<<<4096,256>>>(n,2.0,x,y);
```

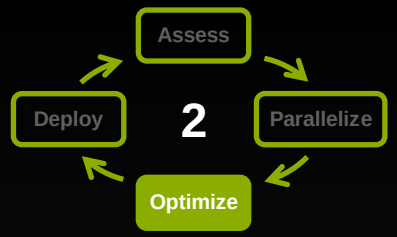
developer.nvidia.com/cuda-toolkit




# APOD CASE STUDY

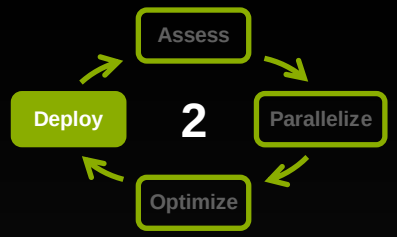
## Round 2: Optimize

# Optimizing OpenACC




```
!$acc data copy(u, v)
do t = 1, 1000
  !$acc kernels
  u(:,:) = u(:,:) + dt  * v(:,:)

  do y=2, ny-1
      do x=2,nx-1
                v(x,y) = v(x,y) + dt * c * …
      end do
   end do
   !$acc end kernels
   !$acc update host(u(1:nx/4,1:2))
   call BoundaryCondition(u)
   !$acc update device(u(1:nx/4, 1:2)
end do
!$acc end data
```

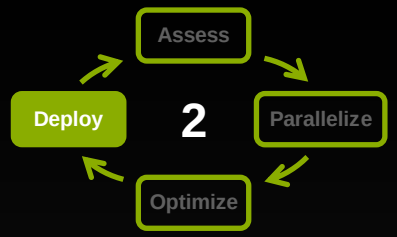
- Usually this means adding some extra directives to give the compiler more information
- This is an entire session:
  - S3019 - Optimizing OpenACC Codes
  - S3533 - Hands-on Lab: OpenACC Optimization




# APOD CASE STUDY

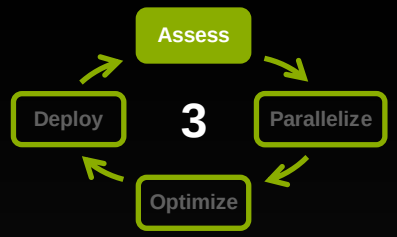
## Round 2: Deploy

# Deploy




- We've removed (or reduced) a bottleneck
- Our app is now faster while remaining fully functional*
- Let's take advantage of that!

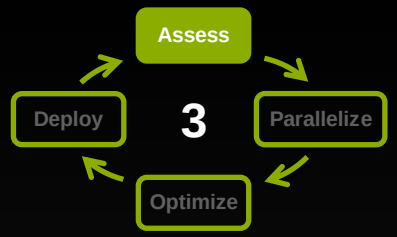
- *Don't forget to check correctness at every step




# APOD CASE STUDY

## Round 3: Assess

# Assess



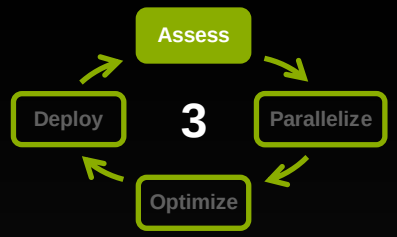

| | Function Name | Module ID | Function ID | Count | Device % | Device Time (µs) | Min (µs) | Avg (µs) | Max (µs) | Context ID |
|---|---|---|---|---|---|---|---|---|---|---|
| 1 | spmv_kernel_v0<int=256> | 8 | 3 | 59 | 35.96 | 457,088.169 | 7,700.182 | 7,747.257 | 7,809.367 | 1 |
| 2 | jacobi_smooth_kernel_v0<int=256> | 6 | 4 | 29 | 1.49 | 18,899.315 | | | 653.224 | 1 |
| 3 | dot_kernel_v0<int=256> | 5 | 2 | 58 | 0.36 | 4,553.489 | | | | 1 |
| 4 | axpbypcz_kernel<int=256> | 5 | 5 | 28 | 0.35 | 4,442.227 | | | 159.394 | 1 |
| 5 | l2_norm_kernel_v0<int=256> | 7 | 1 | 30 | 0.30 | 3,793.612 | | | 128.098 | 1 |
| 6 | axpby_kernel<int=256> | 5 | 4 | 31 | 0.29 | 3,741.159 | | | | 1 |
| 7 | jacobi_invert_diag_kernel_v0<int=256> | 6 | 1 | 1 | 0.06 | 783.209 | | | 783.209 | 1 |
| 8 | reduce_kernel<int=256> | 5 | 1 | 58 | 0.03 | 366.597 | | | 7.168 | 1 |
| 9 | reduce_l2_norm_kernel<int=256> | 7 | 2 | 30 | 0.02 | 291.490 | | | 10.560 | 1 |

Device Time (µs)
18,899.315
4,553.489
4,442.227
3,793.612
3,741.159
783.209
366.597
291.490

~93%

- Finally got rid of those other bottlenecks
- Time to go dig in to that top kernel

# Assess



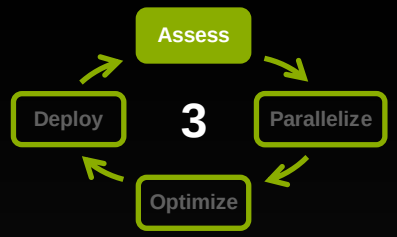

| | Function Name | Module ID | Function ID | Count | Device % | Device Time (µs) | Min (µs) | Avg (µs) | Max (µs) | Context ID |
|---|---|---|---|---|---|---|---|---|---|---|
| 1 | spmv_kernel_v0<int=256> | 8 | 3 | 59 | 35.96 | 457,088.169 | 7,700.182 | 7,747.257 | 7,809.367 | 1 |
| 2 | jacobi_smooth_kernel_v0<int=256> | 6 | 4 | 29 | 1.49 | 18,899.315 | | | 653.224 | 1 |
| 3 | dot_kernel_v0<int=256> | 5 | 2 | 58 | 0.36 | 4,553.489 | | | | 1 |
| 4 | axpbypcz_kernel<int=256> | 5 | 5 | 28 | 0.35 | 4,442.227 | | | 159.394 | 1 |
| 5 | l2_norm_kernel_v0<int=256> | 7 | 1 | 30 | 0.30 | 3,793.612 | | | 128.098 | 1 |
| 6 | axpby_kernel<int=256> | 5 | 4 | 31 | 0.29 | 3,741.159 | | | | 1 |
| 7 | jacobi_invert_diag_kernel_v0<int=256> | 6 | 1 | 1 | 0.06 | 783.209 | | | 783.209 | 1 |
| 8 | reduce_kernel<int=256> | 5 | 1 | 58 | 0.03 | 366.597 | | | 7.168 | 1 |
| 9 | reduce_l2_norm_kernel<int=256> | 7 | 2 | 30 | 0.02 | 291.490 | | | 10.560 | 1 |

Device Time (µs)
18,899.315
4,553.489
4,442.227
3,793.612
3,741.159
783.209
366.597
291.490

~93%

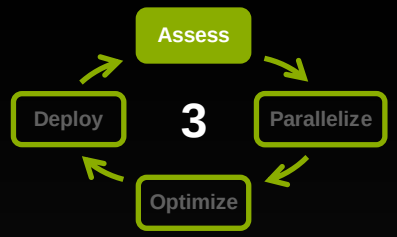
- What percent of our total time does this represent?
- How much can we improve it? What is the “speed of light”?
- How much will this improve our overall performance?

# Assess




- Let's investigate…
  - Strong scaling and Amdahl's Law
  - Weak scaling and Gustafson's Law
  - Expected perf limiters: Bandwidth? Computation? Latency?

# Assess: Understanding Scaling



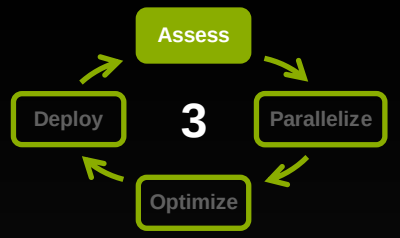

## Strong Scaling

- A measure of how, for fixed overall problem size, the time to solution decreases as more processors are added to a system
- Linear strong scaling: speedup achieved is equal to number of processors used

- Amdahl's Law:

$$S = \frac{1}{(1-P) + \frac{P}{N}} \approx \frac{1}{(1-P)}$$

# Assess: Understanding Scaling



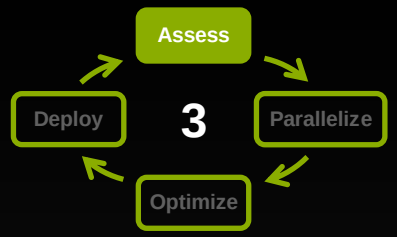

## Weak Scaling

- A measure of how time to solution changes as more processors are added with fixed problem size *per processor*
- Linear weak scaling: overall problem size increases as num. of processors increases, but execution time remains constant

- Gustafson's Law:

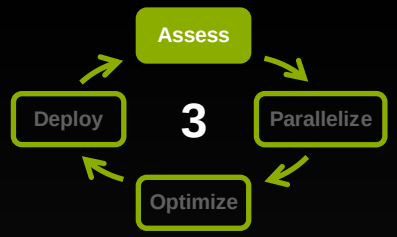
$$S = N + (1 - P)(1 - N)$$

# Assess: Applying Strong and Weak Scaling




- Understanding which type of scaling is most applicable is an important part of estimating speedup:
  - Sometimes problem size will remain constant
  - Other times problem size will grow to fill the available processors
- Apply either Amdahl's or Gustafson's Law to determine an upper bound for the speedup

# Assess: Applying Strong Scaling



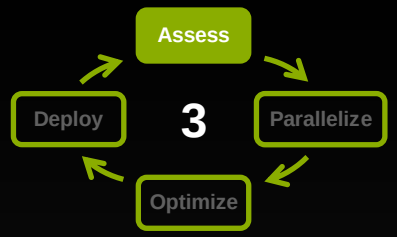

- Recall that in this case we are wanting to optimize an existing kernel with a pre-determined workload

- That's strong scaling, so Amdahl's Law will determine the maximum speedup

| | Function Name | Module ID | Function ID | Count | Device % | Device Time (µs) | Min (µs) | Avg (µs) | Max (µs) | Context ID |
|---|---|---|---|---|---|---|---|---|---|---|
| 1 | spmv_kernel_v0<int=256> | 8 | 3 | 59 | 35.96 | 457,088.169 | 7,700.182 | 7,747.257 | 7,809.367 | 1 |
| 2 | jacobi_smooth_kernel_v0<int=256> | 6 | 4 | 29 | 1.49 | 18,899.315 | | | 653.224 | 1 |
| 3 | [illegible] | 5 | 2 | 58 | 0.36 | 4,553.489 | | | [illegible] | 1 |
| 4 | axpbypcz_kernel<int=256> | 5 | 5 | 28 | 0.35 | [illegible] | | | 159.394 | 1 |
| 5 | l2_norm_kernel_v0<int=256> | 7 | 1 | 30 | 0.30 | [illegible] | | | 128.098 | 1 |
| 6 | axpby_kernel<int=256> | 5 | 4 | 31 | 0.29 | [illegible] | | | [illegible] | 1 |
| 7 | jacobi_invert_diag_kernel_v0<int=256> | 6 | 1 | 1 | 0.06 | 783.209 | | 4,553.489 | 783.209 | 1 |
| 8 | reduce_kernel<int=256> | 5 | 1 | 58 | 0.03 | 366.597 | | 4,442.227 | 7.168 | 1 |
| 9 | reduce_l2_norm_kernel<int=256> | 7 | 2 | 30 | 0.02 | 291.490 | | 3,793.612 | 10.560 | 1 |
| | | | | | | | | 3,741.159 | | |
| | | | | | | | | 783.209 | | |
| | | | | | | | | 366.597 | | |
| | | | | | | | | 291.490 | | |

~93%

# Assess: Applying Strong Scaling

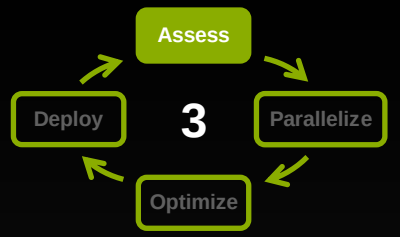



- Now that we've removed the other bottlenecks, our kernel is ~93% of total time
- Speedup $S = \frac{1}{(1-P)+\frac{P}{S_P}}$ ($S_P$ = speedup in parallel part)
- In the limit when $S_P$ is huge, $S$ will approach $\frac{1}{1-0.93} \approx 14.3\times$
- In practice, it will be less than that depending on the $S_P$ achieved

| | Function Name | Module ID | Function ID | Count | Device % | Device Time (µs) | Min (µs) | Avg (µs) | Max (µs) | Context ID |
|---|---|---|---|---|---|---|---|---|---|---|
| 1 | spmv_kernel_v0<int=256> | 8 | 3 | 59 | 35.96 | 457,088.169 | 7,700.182 | 7,747.257 | 7,809.367 | 1 |
| 2 | jacobi_smooth_kernel_v0<int=256> | 6 | 4 | 29 | 1.49 | 18,899.315 | | | 653.224 | 1 |
| 3 | [illegible] | 5 | 2 | 58 | 0.36 | 4,553.489 | | | [illegible] | 1 |
| 4 | axpbypcz_kernel<int=256> | 5 | 5 | 28 | 0.35 | [illegible] | | | 159.394 | 1 |
| 5 | l2_norm_kernel_v0<int=256> | 7 | 1 | 30 | 0.30 | [illegible] | | | 128.098 | 1 |
| 6 | axpby_kernel<int=256> | 5 | 4 | 31 | 0.29 | [illegible] | | | [illegible] | 1 |
| 7 | jacobi_invert_dlag_kernel_v0<int=256> | 6 | 1 | 1 | 0.06 | 783.209 | | 4,553.489 | 783.209 | 1 |
| 8 | reduce_kernel<int=256> | 5 | 1 | 58 | 0.03 | 366.597 | | 4,442.227 | 7.168 | 1 |
| 9 | reduce_l2_norm_kernel<int=256> | 7 | 2 | 30 | 0.02 | 291.490 | | 3,793.612 | 10.560 | 1 |
| | | | | | | | | 3,741.159 | | |
| | | | | | | | | 783.209 | | |
| | | | | | | | | 366.597 | | |
| | | | | | | | | 291.490 | | |

~93%

# Assess: Speed of Light



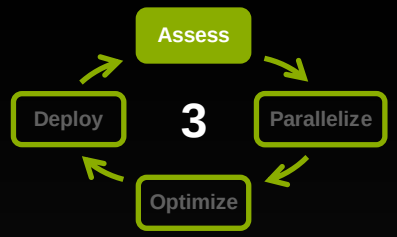

- What's the limiting factor?
  - Memory bandwidth? Compute throughput? Latency?
- For our example kernel, SpMV, we think it should be bandwidth
- We're getting only ~38% of peak bandwidth. If we could get this to 65% of peak, that would mean 1.7× for this kernel, 1.6× overall

$$S = \frac{1}{(1-0.93)+\frac{0.93}{1.7}} \approx 1.6\times$$

| | Function Name | Module ID | Function ID | Count | Device % | Device Time (µs) | Min (µs) | Avg (µs) | Max (µs) | Context ID |
|---|---|---|---|---|---|---|---|---|---|---|
| 1 | spmv_kernel_v0<int=256> | 8 | 3 | 59 | 35.96 | 457,088.169 | 7,700.182 | 7,747.257 | 7,809.367 | 1 |
| 2 | jacobi_smooth_kernel_v0<int=256> | 6 | 4 | 29 | 1.49 | 18,899.315 | | | 653.224 | 1 |
| 3 | [illegible] | 5 | 2 | 58 | 0.36 | 4,553.489 | | | [illegible] | 1 |
| 4 | axpbypcz_kernel<int=256> | 5 | 5 | 28 | 0.35 | [illegible] | | | 159.394 | 1 |
| 5 | l2_norm_kernel_v0<int=256> | 7 | 1 | 30 | 0.30 | [illegible] | | | 128.098 | 1 |
| 6 | axpby_kernel<int=256> | 5 | 4 | 31 | 0.29 | [illegible] | | | [illegible] | 1 |
| 7 | jacobi_invert_diag_kernel_v0<int=256> | 6 | 1 | 1 | 0.06 | 783.209 | | 4,553.489 | 783.209 | 1 |
| 8 | reduce_kernel<int=256> | 5 | 1 | 58 | 0.03 | 366.597 | | 4,442.227 | 7.168 | 1 |
| 9 | reduce_l2_norm_kernel<int=256> | 7 | 2 | 30 | 0.02 | 291.490 | | 3,793.612 | 10.560 | 1 |
| | | | | | | | | 3,741.159 | | |
| | | | | | | | | 783.209 | | |
| | | | | | | | | 366.597 | | |
| | | | | | | | | 291.490 | | |

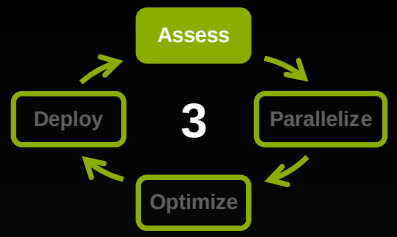
~93%

# Assess: Limiting Factor



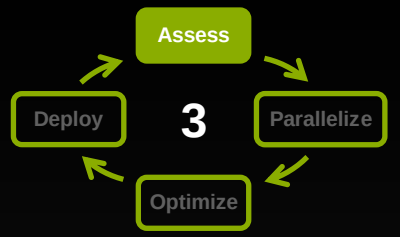

- What's the limiting factor?
  - Memory bandwidth
  - Compute throughput
  - Latency
- Not sure?
  - Get a rough estimate by counting bytes per instruction, compare it to "balanced" peak ratio $\frac{GBytes/sec}{Ginsns/sec}$
  - Profiler will help you determine this

# Assess: Limiting Factor




- Comparing bytes per instr. will give you a guess as to whether you're likely to be bandwidth-bound or instruction-bound
- Comparing actual achieved GB/s vs. theory and achieved Ginstr/s vs. theory will give you an idea of how well you're doing
  - If both are low, then you're probably latency-bound and need to expose more (concurrent) parallelism

# Assess: Limiting Factor

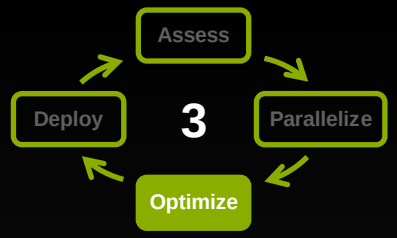



- For our example kernel, our first discovery was that we're latency-limited, not bandwidth, since utilization was so low

- This tells us our first "optimization" step actually needs to be related how we expose (memory-level) parallelism

| | Function Name | Module ID | Function ID | Count | Device % | Device Time (µs) | Min (µs) | Avg (µs) | Max (µs) | Context ID |
|---|---|---|---|---|---|---|---|---|---|---|
| 1 | spmv_kernel_v0<int=256> | 8 | 3 | 59 | 35.96 | 457,088.169 | 7,700.182 | 7,747.257 | 7,809.367 | 1 |
| 2 | jacobi_smooth_kernel_v0<int=256> | 6 | 4 | 29 | 1.49 | 18,899.315 | [illegible] | [illegible] | 653.224 | 1 |
| 3 | [illegible] | 5 | 2 | 58 | 0.36 | 4,553.489 | [illegible] | [illegible] | [illegible] | 1 |
| 4 | axpbypcz_kernel<int=256> | 5 | 5 | 28 | 0.35 | [illegible] | [illegible] | [illegible] | 159.394 | 1 |
| 5 | l2_norm_kernel_v0<int=256> | 7 | 1 | 30 | 0.30 | [illegible] | [illegible] | [illegible] | 128.098 | 1 |
| 6 | axpby_kernel<int=256> | 5 | 4 | 31 | 0.29 | [illegible] | [illegible] | [illegible] | [illegible] | 1 |
| 7 | jacobi_invert_diag_kernel_v0<int=256> | 6 | 1 | 1 | 0.06 | 783.209 | [illegible] | 4,553.489 | 783.209 | 1 |
| 8 | reduce_kernel<int=256> | 5 | 1 | 58 | 0.03 | 366.597 | [illegible] | 4,442.227 | 7.168 | 1 |
| 9 | reduce_l2_norm_kernel<int=256> | 7 | 2 | 30 | 0.02 | 291.490 | [illegible] | 3,793.612 | 10.560 | 1 |
| | | | | | | | | 3,741.159 | | |
| | | | | | | | | 783.209 | | |
| | | | | | | | | 366.597 | | |
| | | | | | | | | 291.490 | | |

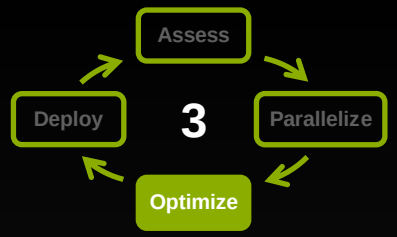
~93%




# APOD CASE STUDY

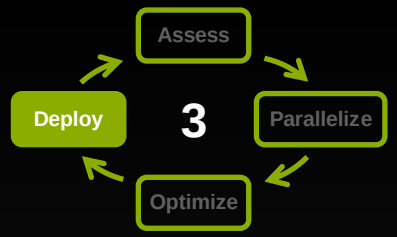
## Round 3: Optimize

# Optimize




- Our optimization efforts should all be profiler-guided
- The tricks of the trade for this can fill an entire session
  - S3011 - Case Studies and Optimization Using Nsight VSE
  - S3535 - Hands-on Lab: CUDA Application Optimization Using Nsight VSE

  - S3046 - Performance Optimization Strategies for GPU Applications
  - S3528 - Hands-on Lab: CUDA Application Optimization Using Nsight EE




# APOD CASE STUDY

## Round 3: Deploy

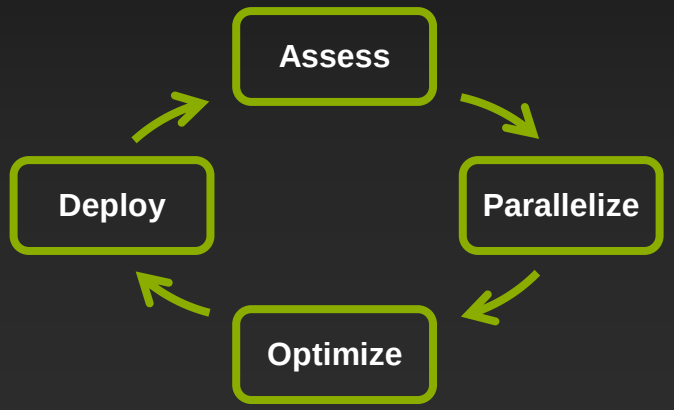
# HIGH-PRODUCTIVITY CUDA:

## Wrap-up

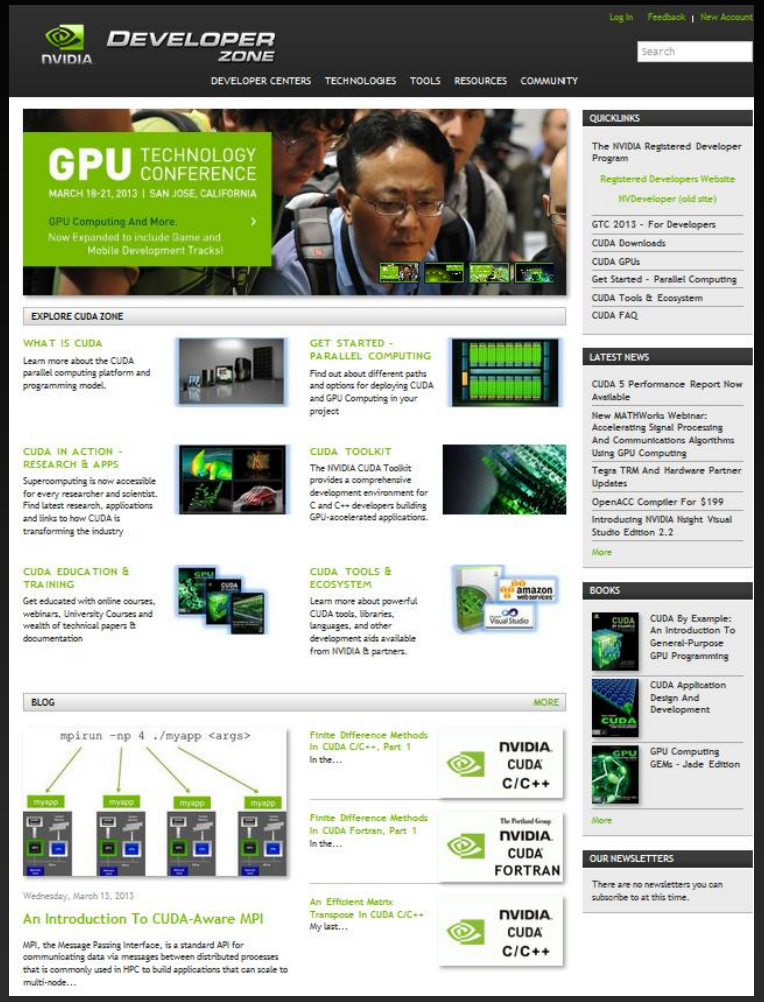
# High-Productivity CUDA Programming

**Recap:**

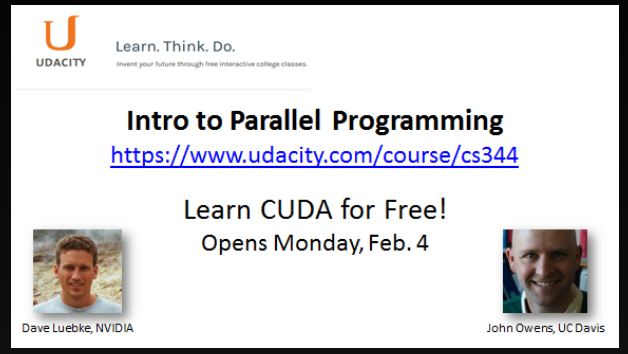
- Know the tools at our disposal
- Use them wisely
- Develop systematically with APOD



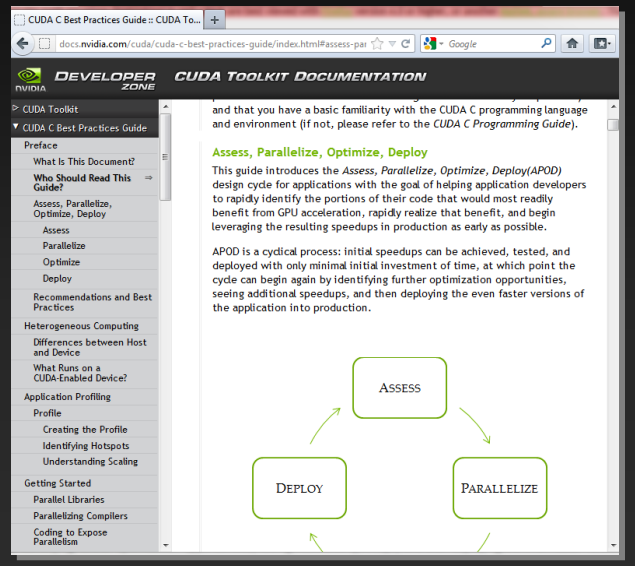
# Online Resources




developer.nvidia.com

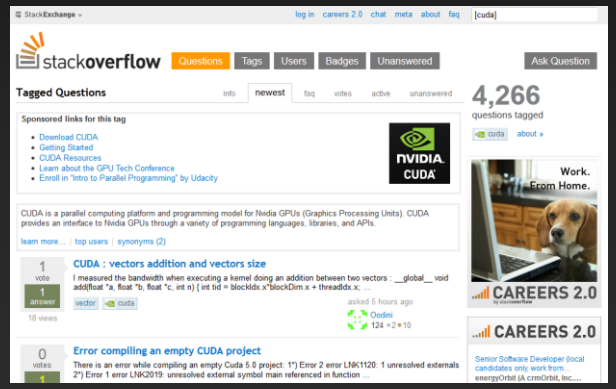



www.udacity.com




docs.nvidia.com


devtalk.nvidia.com




www.stackoverflow.com