# NVIDIA GPU Libraries

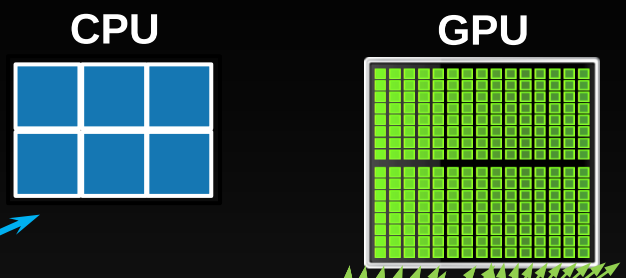
Peter Messmer

# 3 Ways to Accelerate Applications

Applications

| Libraries | OpenACC Directives | Programming Languages |
|---|---|---|
| "Drop-in" Acceleration | Easily Accelerate Applications | Maximum Flexibility |

# OpenACC Directives




```
Program myscience
 ... serial code ...
!$acc kernels
 do k = 1,n1
   do i = 1,n2
      ... parallel code ...
   enddo
  enddo
!$acc end kernels
 ...
End Program myscience
```

OpenACC Compiler Hint

**Your original Fortran or C code**

Simple Compiler hints

Compiler Parallelizes code

Works on many-core GPUs & multicore CPUs

# C for CUDA : C++ with a few keywords

```c
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];
}
// Invoke serial SAXPY kernel
saxpy_serial(n, 2.0, x, y);
```

***Standard C Code***

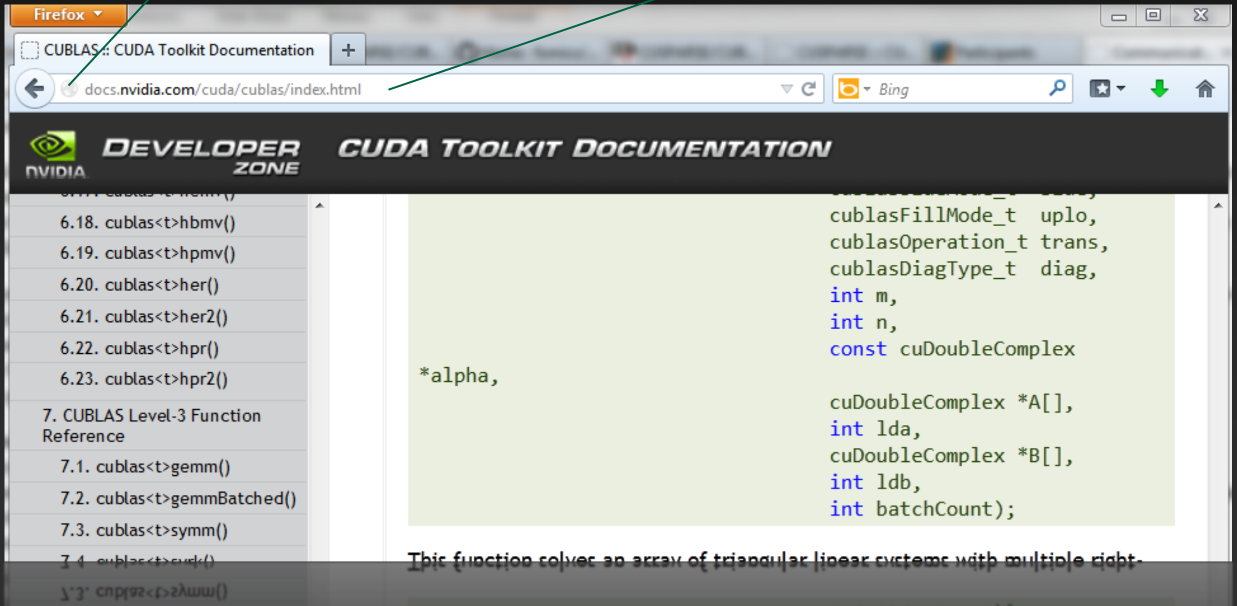
```c
__global__ void saxpy_parallel(int n, float a, float *x, float *y)
{
    int i = blockIdx.x*blockDim.x + threadIdx.x;
    if (i < n)  y[i] = a*x[i] + y[i];
}
// Invoke parallel SAXPY kernel with 256 threads/block
int nblocks = (n + 255) / 256;
saxpy_parallel<<<nblocks, 256>>>(n, 2.0, x, y);
```

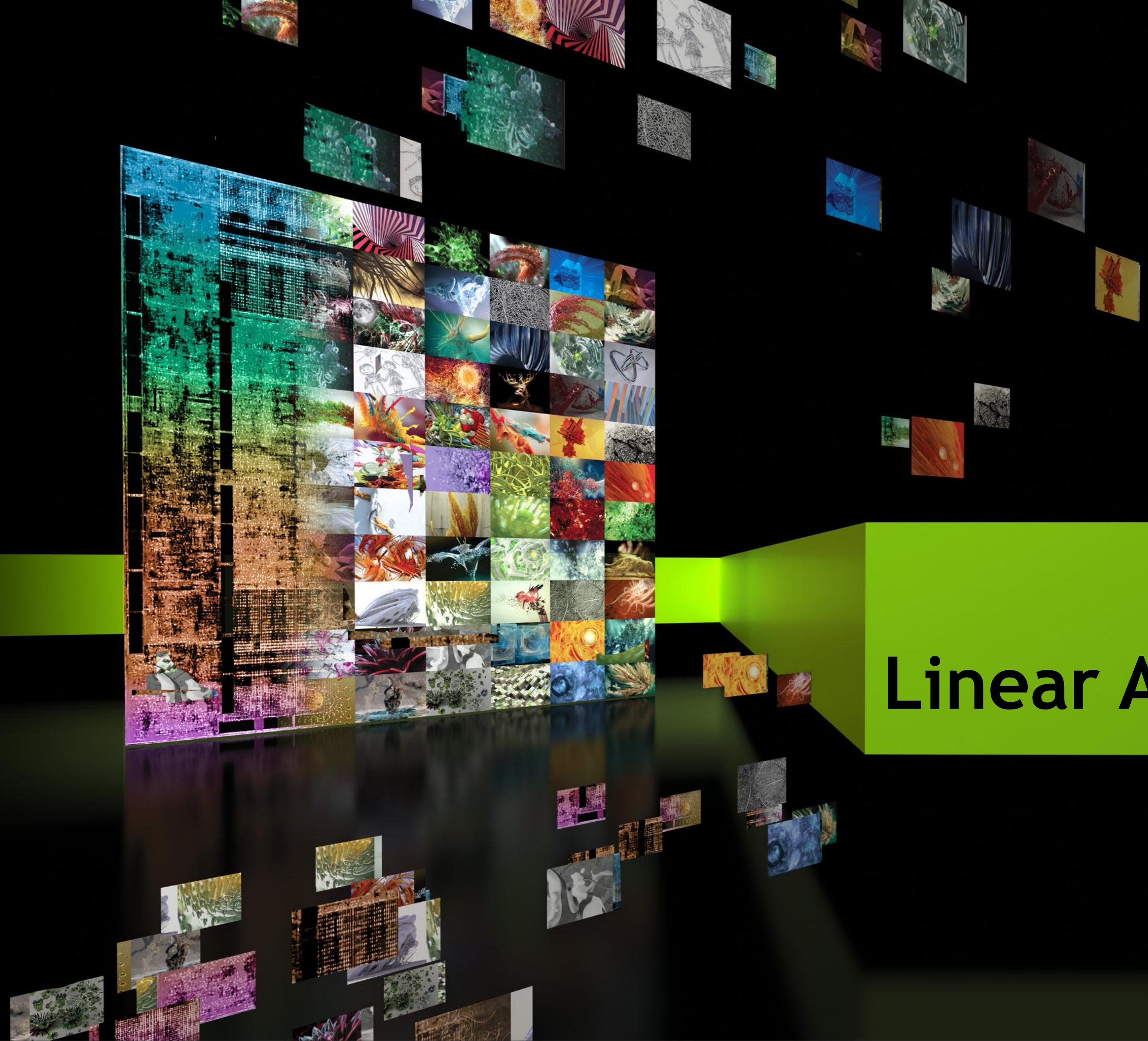
***Parallel C Code***

# CUDA Libraries: Outline

- Introduction of Libraries
  - Linear Algebra: cuBLAS, cuSPARSE
  - Signal Processing: cuFFT, NPP
  - Random Numbers: cuRAND
  - C++ development: Thrust
  - Basic math functions: math.h
- Tools
  - Debugging and profiling
- Software engineering
  - If you develop libraries..
- Hands-on Exercise

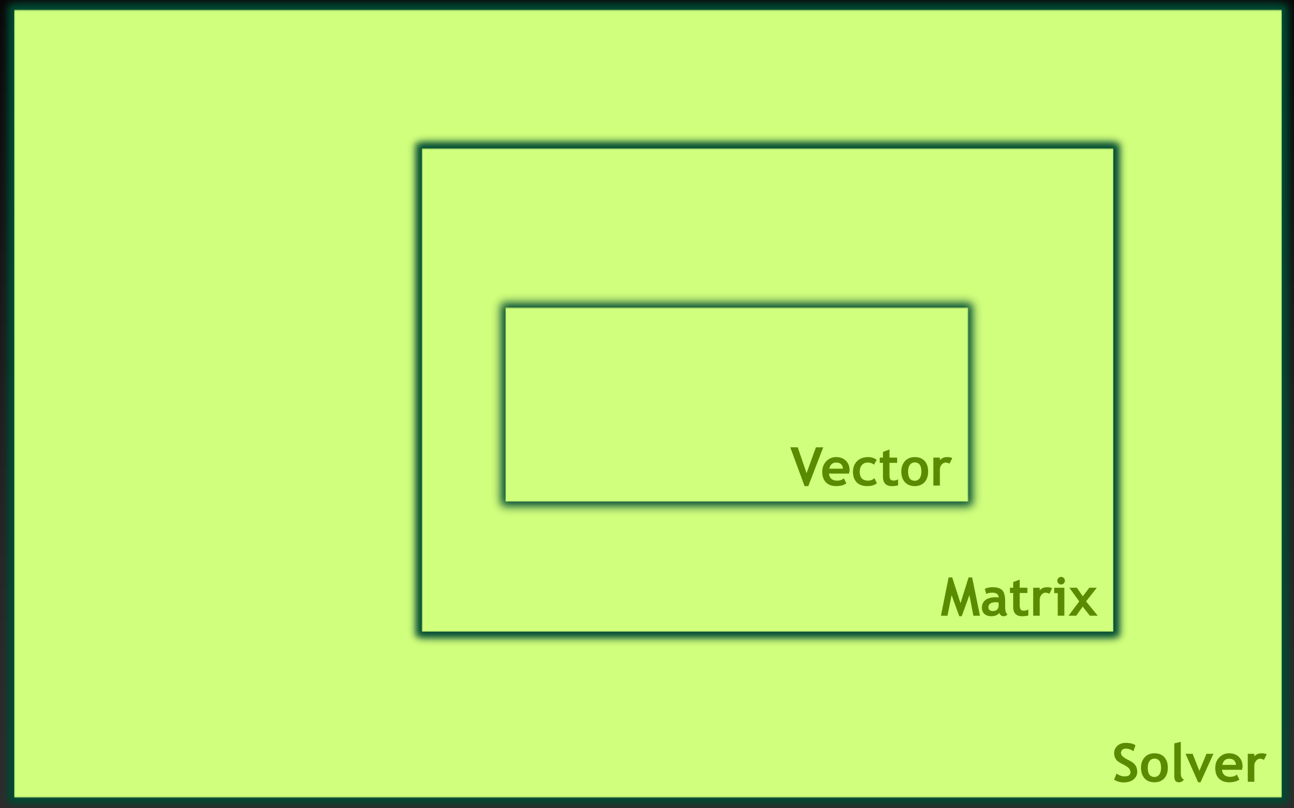
**http://docs.nvidia.com**



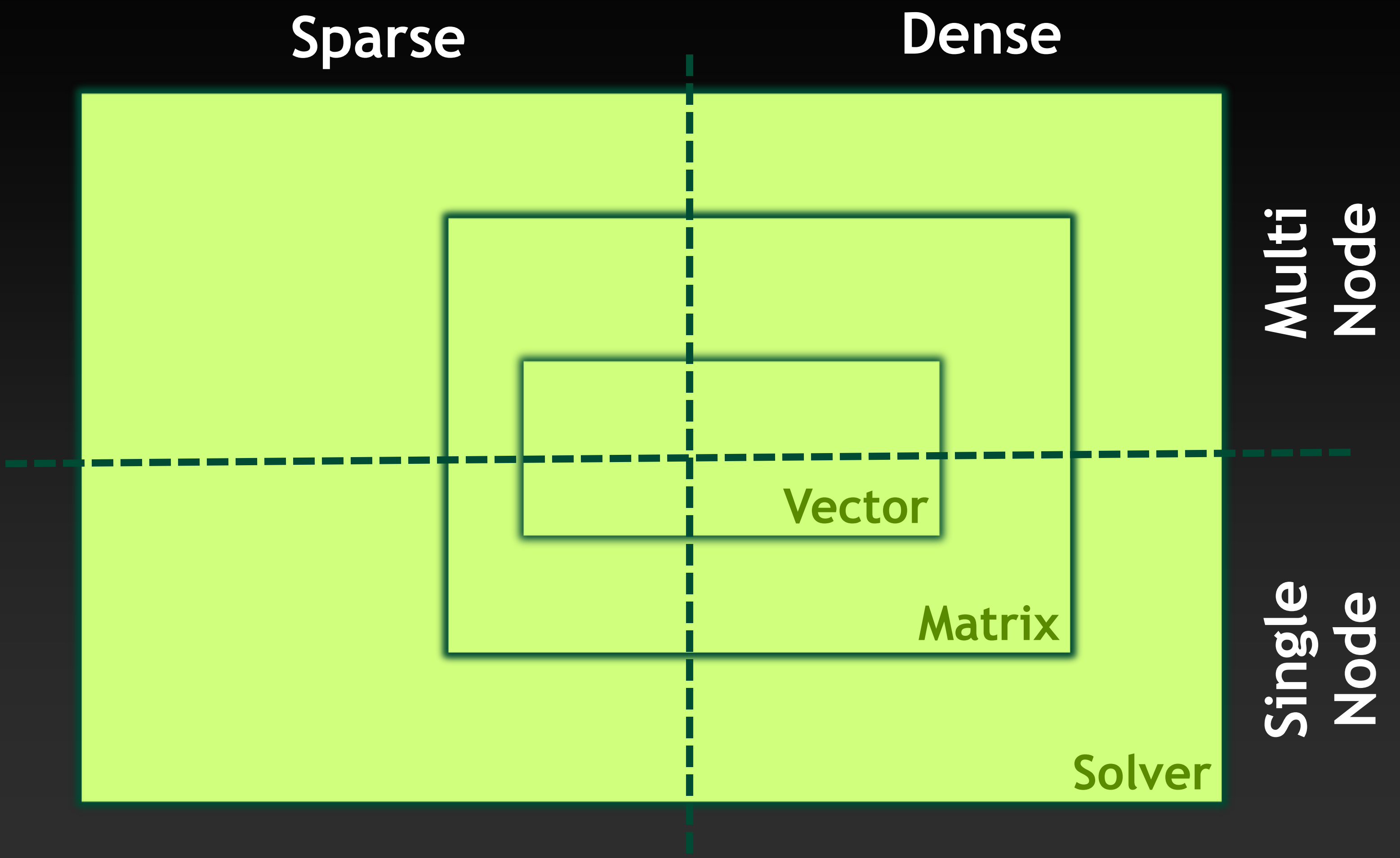


# Linear Algebra

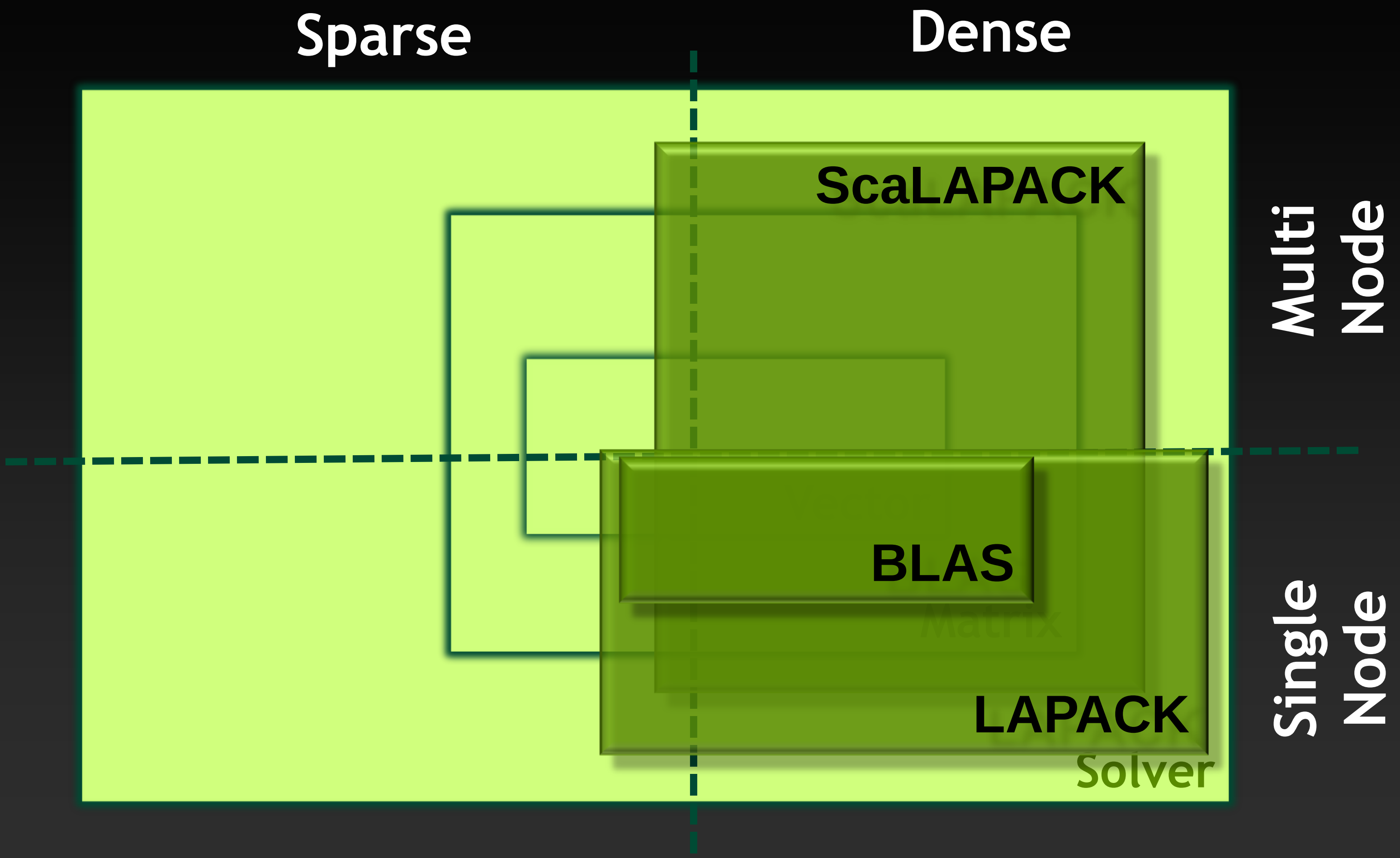
# A Birds Eye View on Linear Algebra



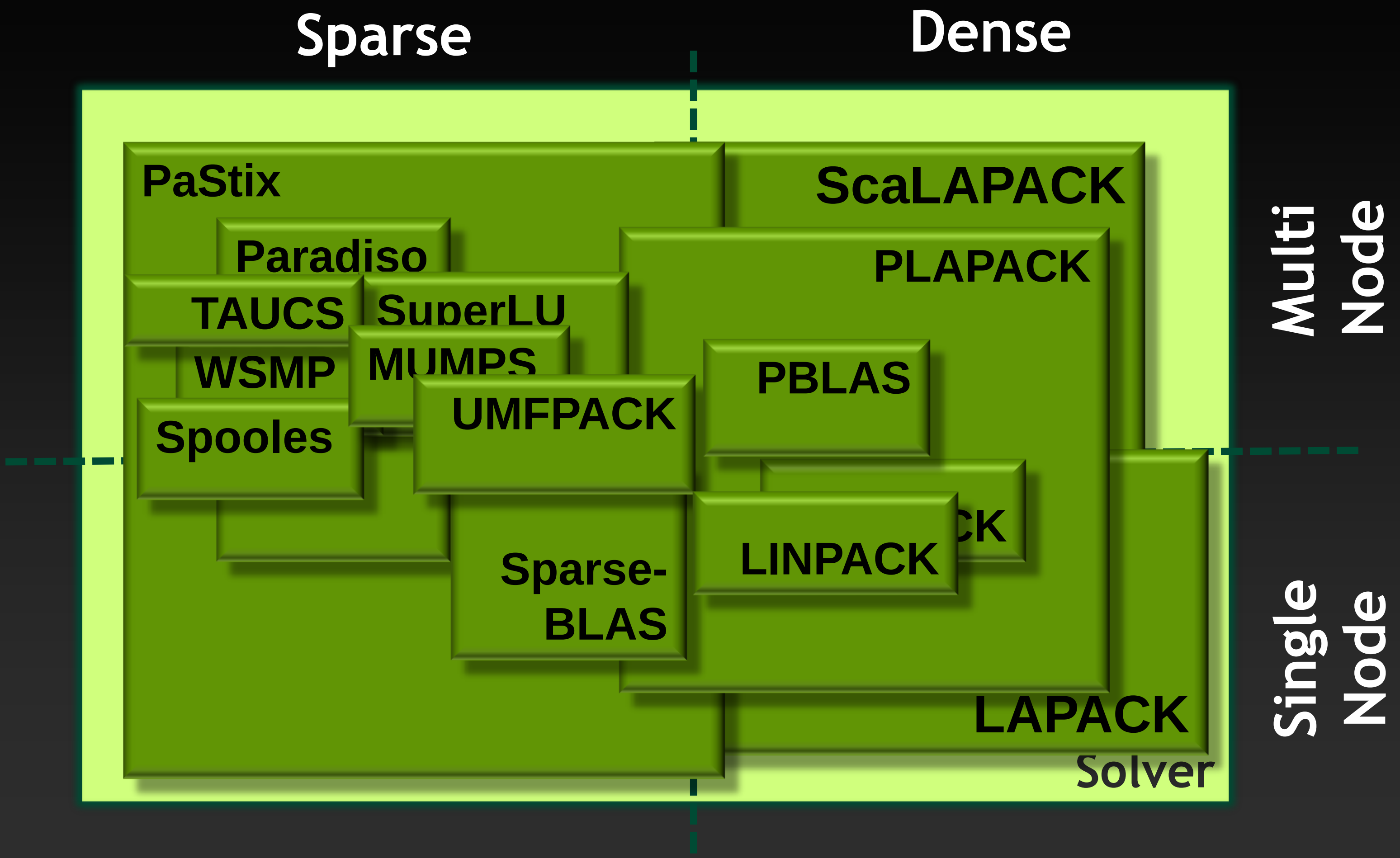
# A Birds Eye View on Linear Algebra



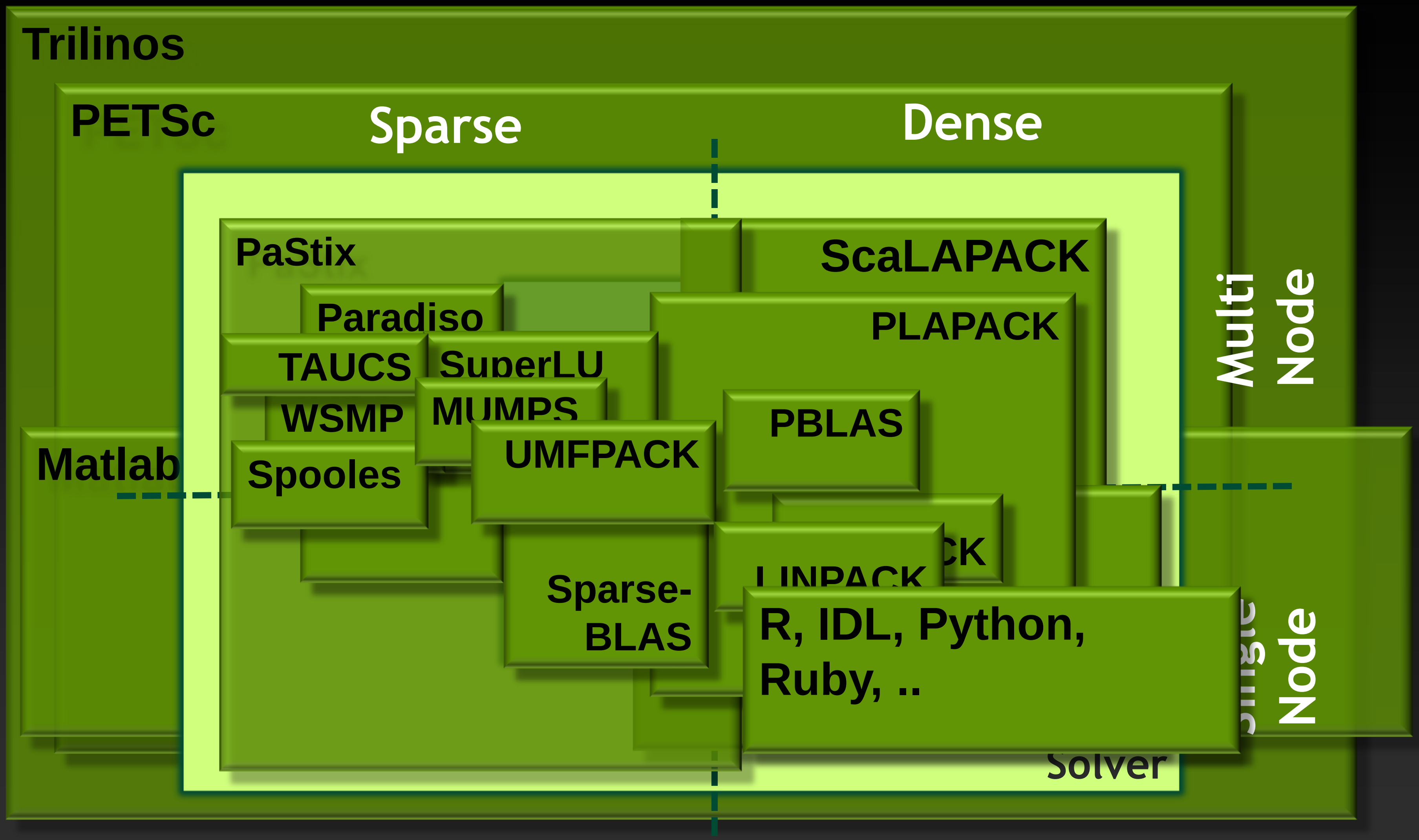
# Sometimes it seems as if there's only three …



# .. but there is more …

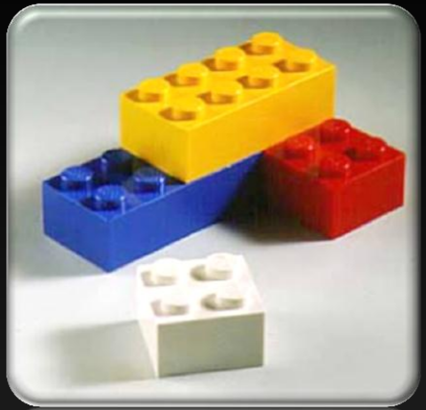


# ... and even more ..



# NVIDIA CUDA Library Approach

- Provide basic building blocks
- Make them easy to use
- Make them fast



- Provides a quick path to GPU acceleration
- Enables developers to focus on their “secret sauce”
- Ideal for applications that use CPU libraries

# Drop-In Acceleration

```
int N = 1 << 20;




// Perform SAXPY on 1M elements: y[]=a*x[]+y[]
saxpy(N, 2.0, d_x, 1, d_y, 1);
```

# Drop-In Acceleration (Step 1)

```
int N = 1 << 20;




// Perform SAXPY on 1M elements: d_y[]=a*d_x[]+d_y[]
cublasSaxpy(N, 2.0, d_x, 1, d_y, 1);
```

Add “cublas” prefix and use device variables

# Drop-In Acceleration (Step 2)

```
int N = 1 << 20;
cublasInit();

// Perform SAXPY on 1M elements: d_y[]=a*d_x[]+d_y[]
cublasSaxpy(N, 2.0, d_x, 1, d_y, 1);

cublasShutdown();
```

Initialize CUBLAS

Shut down CUBLAS

# Drop-In Acceleration (Step 3)

```
int N = 1 << 20;
cublasInit();
cublasAlloc(N, sizeof(float), (void**)&d_x);
cublasAlloc(N, sizeof(float), (void*)&d_y);
```

Allocate device vectors

```
// Perform SAXPY on 1M elements: d_y[]=a*d_x[]+d_y[]
cublasSaxpy(N, 2.0, d_x, 1, d_y, 1);
```

```
cublasFree(d_x);
cublasFree(d_y);
cublasShutdown();
```

Deallocate device vectors

# Drop-In Acceleration (Step 4)

```
int N = 1 << 20;
cublasInit();
cublasAlloc(N, sizeof(float), (void**)&d_x);
cublasAlloc(N, sizeof(float), (void**)&d_y);

cublasSetVector(N, sizeof(x[0]), x, 1, d_x, 1);
cublasSetVector(N, sizeof(y[0]), y, 1, d_y, 1);

// Perform SAXPY on 1M elements: d_y[]=a*d_x[]+d_y[]
cublasSaxpy(N, 2.0, d_x, 1, d_y, 1);

cublasGetVector(N, sizeof(y[0]), d_y, 1, y, 1);

cublasFree(d_x);
cublasFree(d_y);
cublasShutdown();
```

Transfer data to GPU

Read data back GPU

# cuBLAS: Legacy and Version 2 Interface

- Legacy Interface
  - Convenient for quick port of legacy code
- Version 2 Interface
  - Reduces data transfer for complex algorithms
    - Return values on CPU or GPU
    - Scalar arguments passed by reference
  - Support for streams and multithreaded environment
  - Batching of key routines

# Version 2 Interface helps reducing memory transfers

Index transferred to CPU, CPU needs vector elements for scale factor

- Legacy Interface

```
idx = cublasIsamax(n, d_column, 1);
err = cublasSscal(n, 1./d_column[idx], row, 1);
```

# Version 2 Interface helps reducing memory transfers

Index transferred to CPU, CPU needs vector elements for scale factor

- Legacy Interface

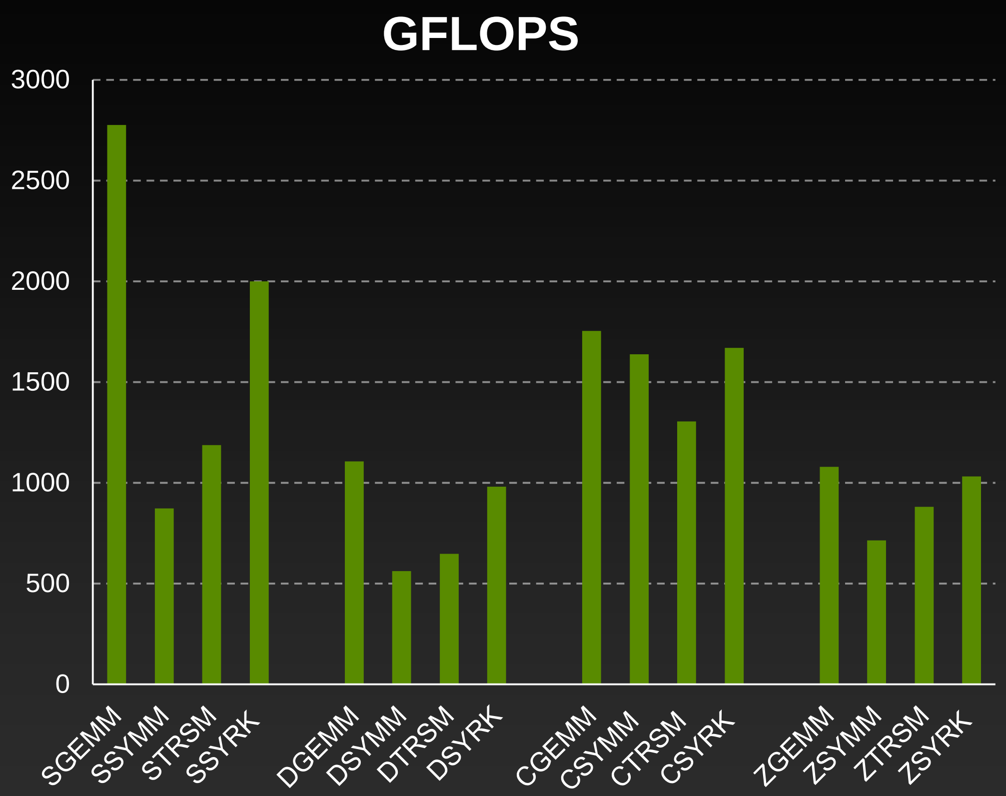
```
idx = cublasIsamax(n, d_column, 1);
err = cublasSscal(n, 1./d_column[idx], row, 1);
```

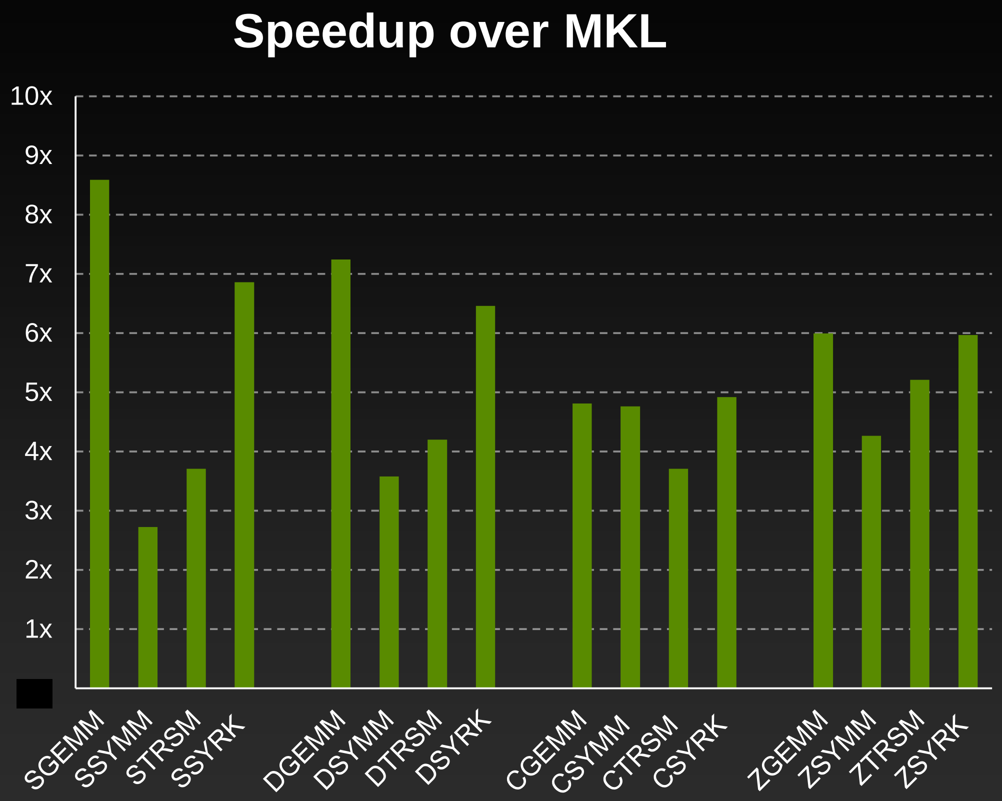
- Version 2 Interface

```
err = cublasIsamax(handle, n, d_column, 1, d_maxIdx);
kernel<<< >>> (d_column, d_maxIdx, d_val);
err = cublasSscal(handle, n, d_val, d_row, 1);
```

All data remains on the GPU

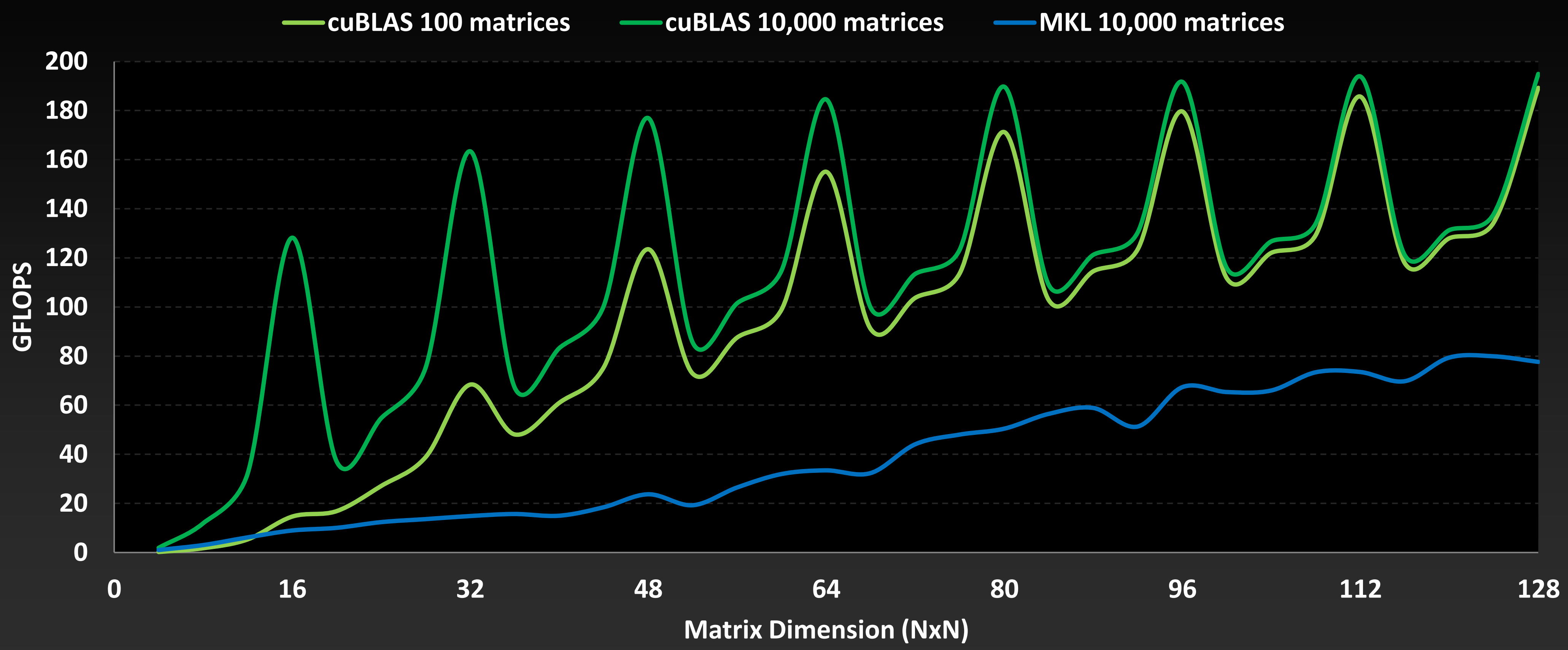
# cuBLAS Level 3: >1 TFLOPS double-precision




- MKL 10.3.6 on Intel SandyBridge E5-2687W @3.10GHz
- CUBLAS 5.0.30 on K20X, input and output data on device

# cuBLAS Batched GEMM API improves performance on batches of small matrices




s

- cuBLAS 4.1 on Tesla M2090, ECC on
- MKL 10.2.3, TYAN FT72-B7015 Xeon x5680 Six-Core @ 3.33 GHz

Performance may vary based on OS version and motherboard configuration

# cuSPARSE Interface

```
mkl_dcsrmv(transa, m, k,
           alpha, descr,
           val, indx, pntrb,
           pntre,
           x, beta, y);
```

```
err = cusparseDcsrmv(hdl,
                     transa, m, k,
                     nnz, alpha, desrc,
                     val, indx, col,
                             x, beta, y);
```

# Different Approaches to Linear Algebra

- CULA tools (dense, sparse)
  - LAPACK based API
  - Solvers, Factorizations, Least Squares, SVD, Eigensolvers
  - Sparse: Krylov solvers, Preconditioners, support for various formats

```
culaSgetrf(M, N, A, LDA, IPIV, INFO)
```

CULA tools

GPU Accelerated Linear Algebra

EM Photonics

- ArrayFire
  - Array container object
  - Solvers, Factorizations, SVD, Eigensolvers

```
array out = lu(A)
```

ArrayFire Matrix Computations

AccelerEyes

# Different Approaches to Linear Algebra (cont.)

- MAGMA
  - LAPACK conforming API
  - Magma BLAS and LAPACK
  - High performance by utilizing both GPU and CPU

```
magma_sgetrf(M, N, A, LDA, IPIV, INFO)
```

MAGMA
Matrix Algebra on GPU and Multicore

ICL

- LibFlame
  - LAPACK compatibility interface
  - Infrastructure for rapid linear algebra algorithm development

```
FLASH_LU_piv(A, p)
```

FLAME Library

UT-Austin

# Different Approaches to Linear Algebra (cont.)

- CUSP
  - Sparse matrix operations
  - Open source
  - Supports COO, CSR, ELL, DIA, hybrid, etc.
  - Solvers, monitors, preconditioners, etc.

```
cusp::krylov::cg(A, x, b);
```

# Toolkits are increasingly supporting GPUs

- PETSc
  - GPU support via extension to Vec and Mat classes
  - Partially dependent on CUSP
  - MPI parallel, GPU accelerated solvers
- Trilinos
  - GPU support in KOKKOS package
  - Used through vector class Tpetra
  - MPI parallel, GPU accelerated solvers

# Signal Processing

# cuFFT

- Interface modeled after FFTW

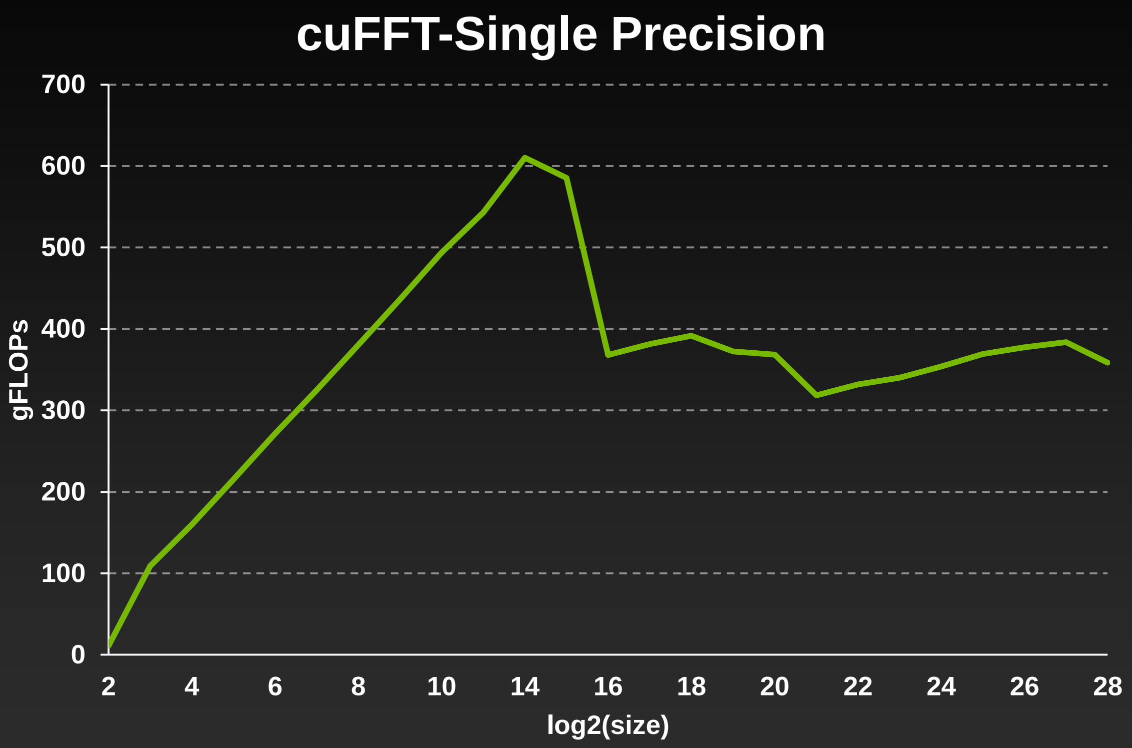
```
fftw_plan PlanA;

fftw_plan_dft_2d(N, M, &PlanA,
         data, data, FFT_FORWARD)

fftw_execute_dft(PlanA, data,
                 data);
```

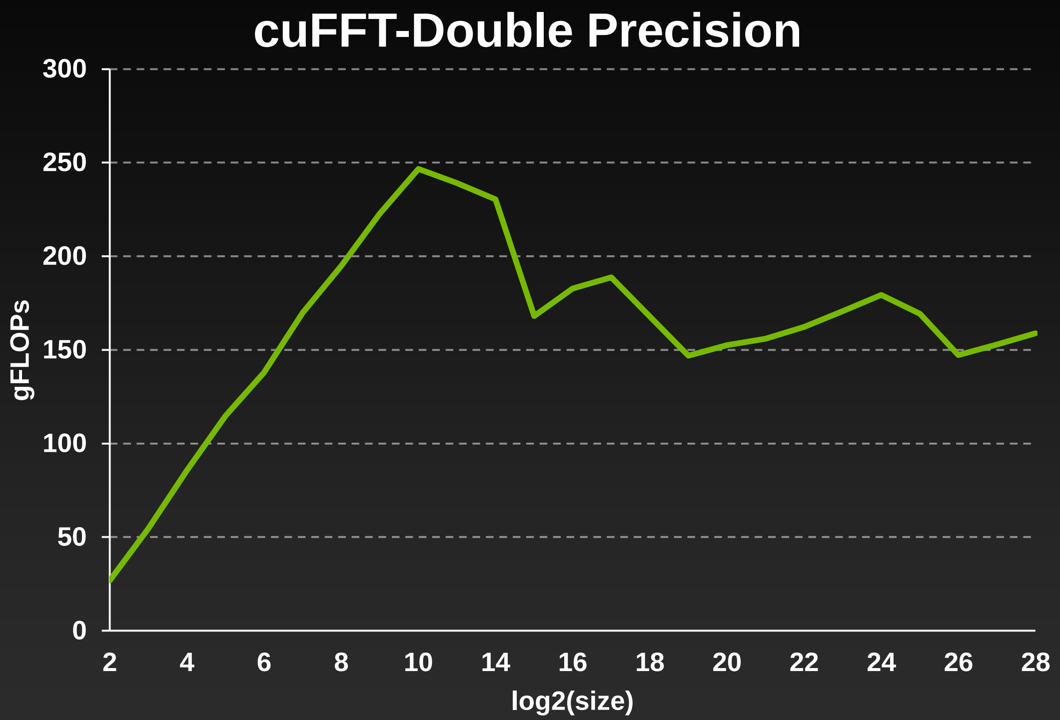
```
cufftPlan2d PlanA;

cufftCreatePlan(N, M, &PlanA,
CUFFT_C2C);

cufftExecC2C(PlanA, d_data, d_data,
             CUFFT_FORWARD);
```

- Supports streams and batching (2 and 3-D, too!) for better performance

# CUFFT: up to 600 GFLOPS

1D used in audio processing and as a foundation for 2D and 3D FFTs

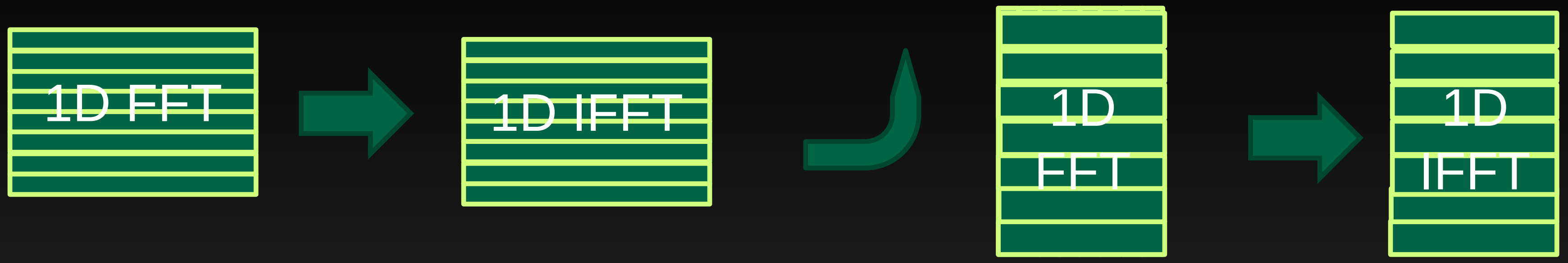



- CUFFT 5.0.30 on K20X, input and output data on device

# cufftPlanMany: Transformation on complex data layouts

- Example: Range-Doppler compression




- No need for explicit transpose with cufftPlanMany
  - Independent input and output strides/internal dimension

```
cufftPlanMany( cufftHandle *plan, int rank, int *n,
        int *inembed, int istride, int idist, // input layout
        int *onembed, int ostride, int odist, // output layout
        cufftType type, int batch)
```

# NPP features a large set of functions

- Arithmetic and Logical Operations
  - Point-by-point ops, clamp, threshold, etc.
- Geometric transformations
  - Rotate, Warp, Interpolate
- Compression
  - jpeg de/compression
- Image processing
  - Filter, histogram, statistics

NVIDIA NPP

# Basic concepts of NPP

- Collection of high-performance GPU processing
  - Non-linear data transforms (point-by-point mult, sqrt, etc.)
  - Support for multi-channel integer and float data
- C API => name disambiguates between data types, flavor

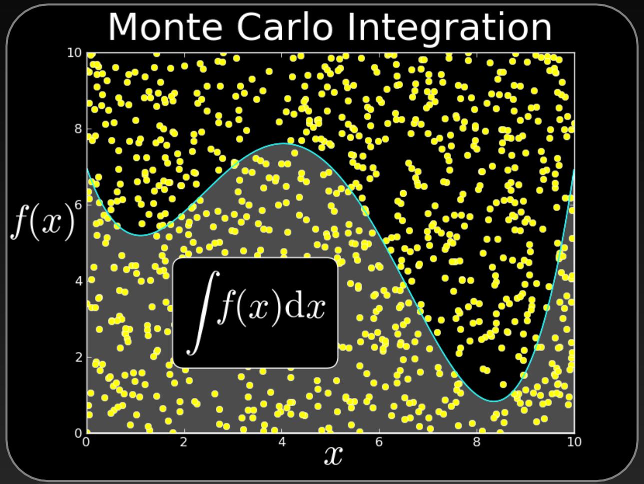
`nppiAdd_32f_C1R (…)`

  - "Add" two single channel ("C1") 32-bit float ("32f") images, possibly masked by a region of interest ("R")

# cuRAND

# Random Number Generation on GPU

- Generating high quality random numbers in parallel is hard
  - Don’t do it yourself, use a library!
- Large suite of generators and distributions
  - XORWOW, MRG323ka, MTGP32, (scrambled) Sobol
  - uniform, normal, log-normal
  - Single and double precision
- Two APIs for cuRAND
  - Called from CPU: Ideal when generating large batches of RNGs on GPU
  - Called from GPU: Ideal when RNGs need to be generated inside a kernel



# cuRAND: Host vs Device API

- CPU API

```
#include "curand.h"
curandCreateGenerator(&gen, CURAND_RNG_PSEUDO_DEFAULT);
curandGenerateUniform(gen, d_data, n);
```

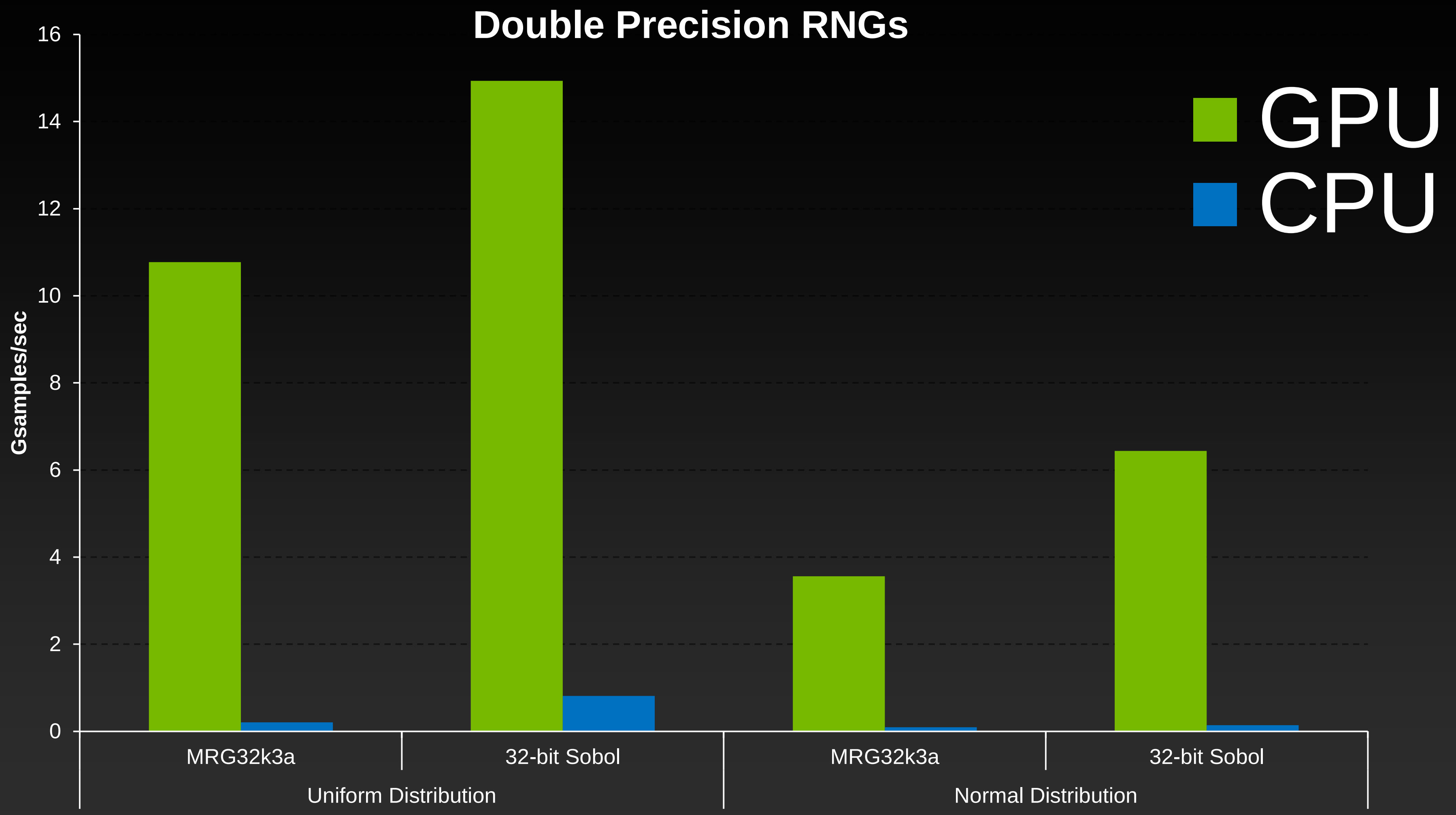
Generate set of random numbers at once

- GPU API

```
#include "curand_kernel.h"
__global__ void generate_kernel(curandState *state) {
      int id = threadIdx.x + blockIdx.x * 64;
      x = curand(&state[id]);
}
```

Generate random numbers per thread

# cuRAND Performance



# Thrust

# Thrust: STL-like CUDA Template Library

- GPU(device) and CPU(host) vector class

```
thrust::host_vector<float> H(10, 1.f);
thrust::device_vector<float> D = H;
```

- Iterators

```
thrust::fill(D.begin(), D.begin()+5, 42.f);
float* raw_ptr = thrust::raw_pointer_cast(D);
```

- Algorithms
  - Sort, reduce, transformation, scan, ..

```
thrust::transform(D1.begin(), D1.end(), D2.begin(), D2.end(),
 thrust::plus<float>());   // D2 = D1 + D2
```

C++ STL Features for CUDA

# math.h

# math.h: C99 floating-point library + extras

CUDA math.h is industry proven, high performance, accurate

- Basic: +, *, /, 1/, sqrt, FMA (all IEEE-754 accurate for float, double, all rounding modes)
- Exponentials: exp, exp2, log, log2, log10, ...
- Trigonometry: sin, cos, tan, asin, acos, atan2, sinh, cosh, asinh, acosh, ...
- Special functions: lgamma, tgamma, erf, erfc
- Utility: fmod, remquo, modf, trunc, round, ceil, floor, fabs, ...
- Extras: rsqrt, rcbrt, exp10, sinpi, sincos, cospi, erfinv, erfcinv, ...

- New in CUDA 5.0
  - sincospi[f]() and normcdf[inv][f]()
  - sin(), cos() and erfcinvf() more accurate and faster
  - Full list of new features and optimizations:

    http://docs.nvidia.com/cuda/cuda-toolkit-release-notes/index.html#math

    http://docs.nvidia.com/cuda/cuda-toolkit-release-notes/index.html#math-performance-improvements

# Explore the CUDA (Libraries) Ecosystem

- CUDA Tools and Ecosystem described in detail on NVIDIA Developer Zone:

  developer.nvidia.com/cuda-tools-ecosystem

# Tools

# Debugging via printf

- printf supported on devices of sm_20 and higher
- Requires inclusion of “stdio.h”

- Caution:
  - Fixed buffer size
  - Flushed under certain circumstances
    - E.g. next time a kernel is launched
  - Not flused by default at end of application
    - Forced eg. via cudaDeviceReset()

# Debugging via cuda-gdb

- Compile with -g -G options
- Use -gencode option

  nvcc -g -G -gencode arch=compute_35,code=sm_35

- run via cuda-gdb myapp

- Determining focus:

  (cuda-gdb) cuda device sm warp lane block thread

- Breakpoint
- (cuda-gdb) break my_file.cu:185

# Debugging via cuda-memcheck

- Useful in case of “unspecified launch failure”
  - Out-of-bound access, memory leaks
- Does not require recompilation

- More precise information if compiled with flags:

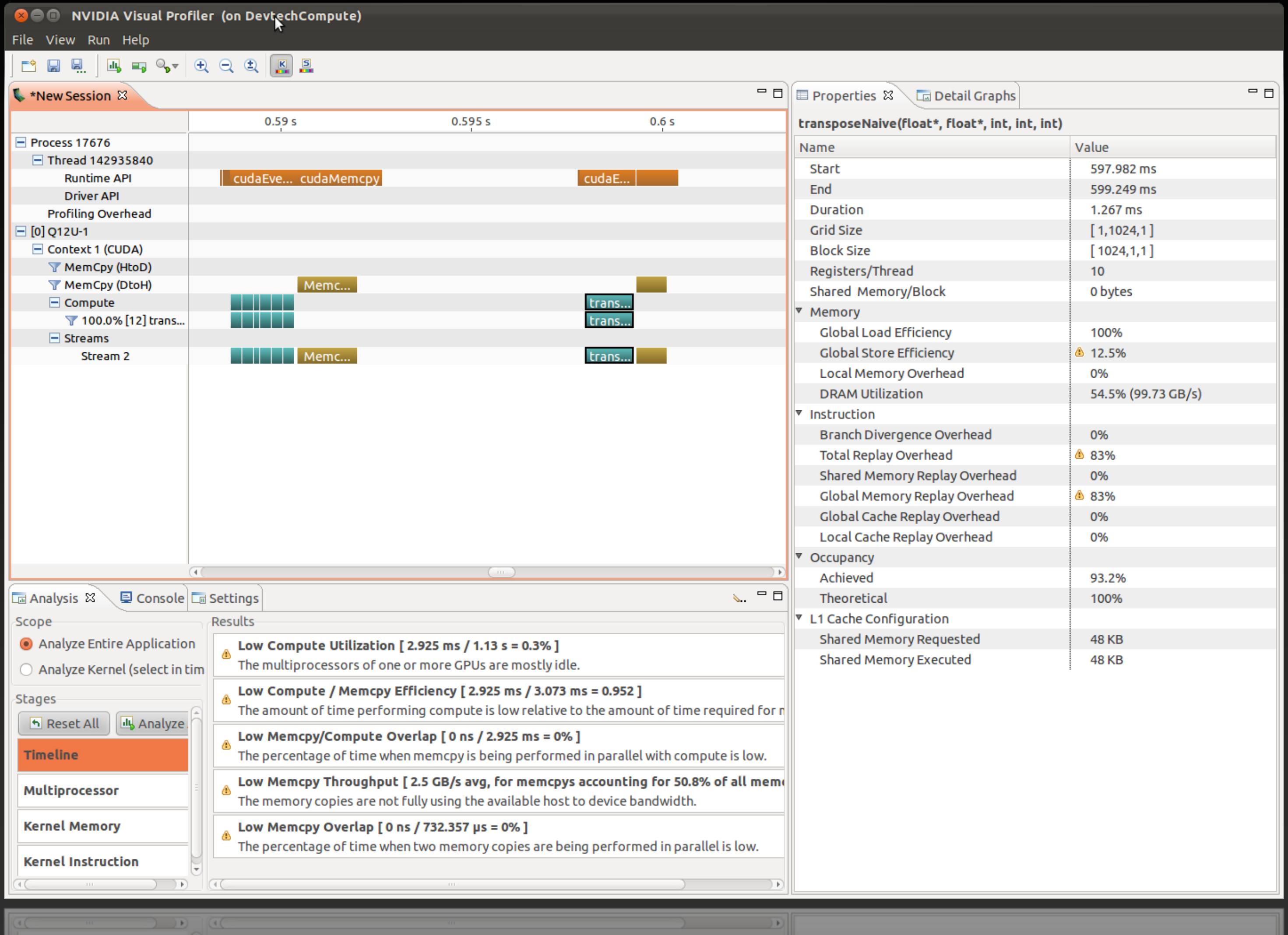
  -G  -lineinfo -rdynamic

- racecheck to detect race conditions

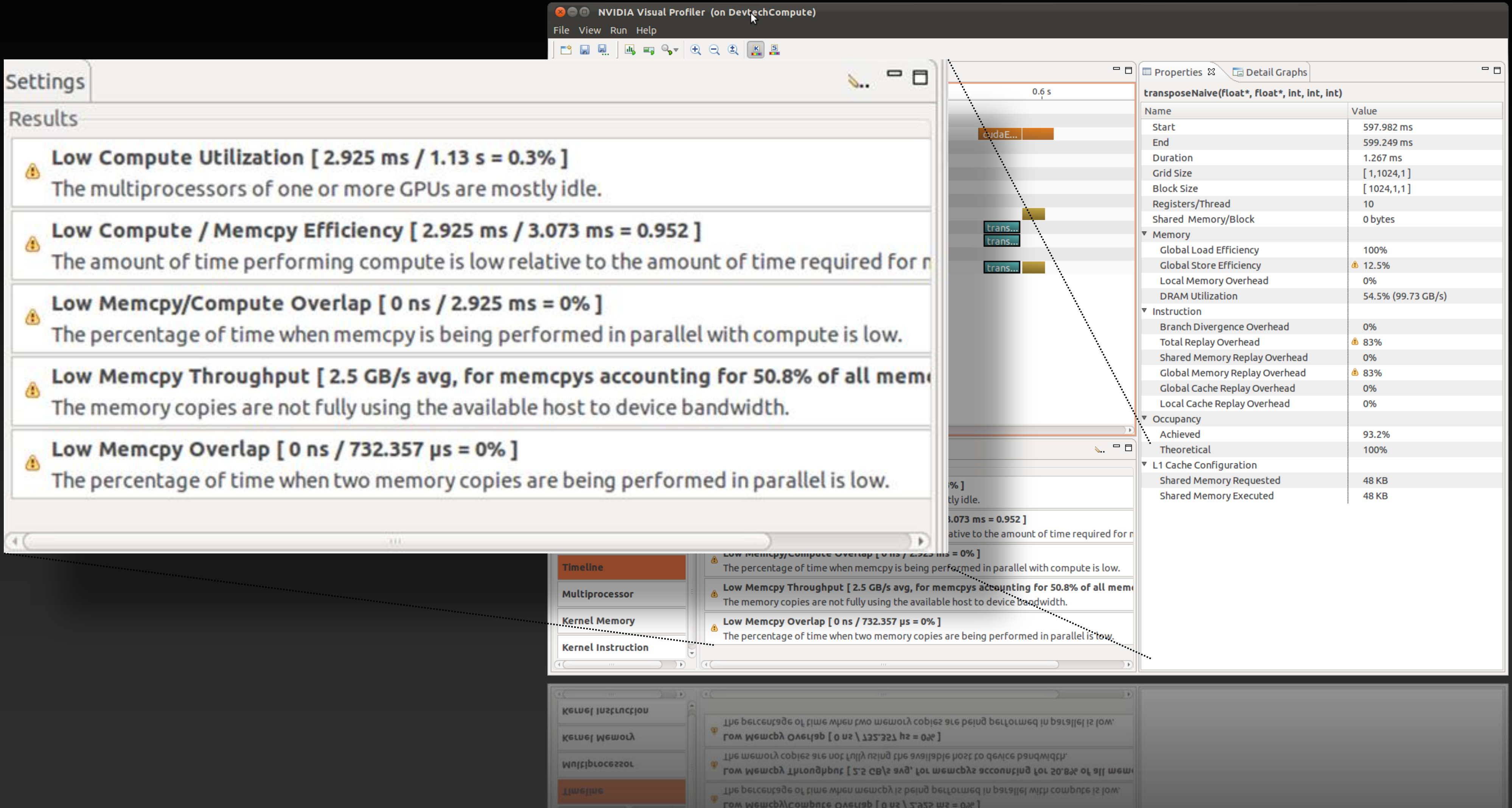
  cuda-memcheck -tool racecheck myapp.x

# NVVP - NVIDIA Visual Profiler

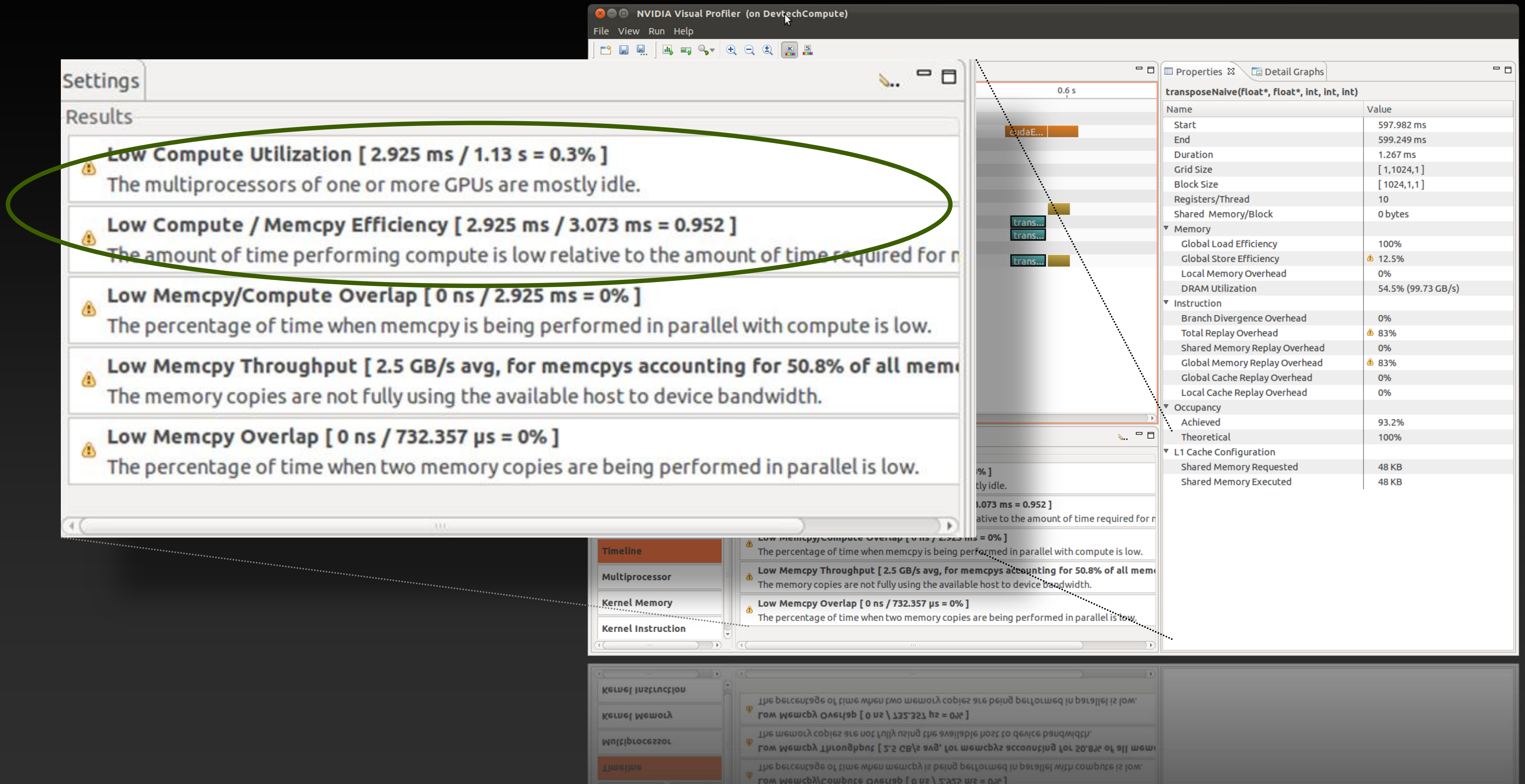
- Application analysis
- Kernel properties



# Application Assessment with NVVP

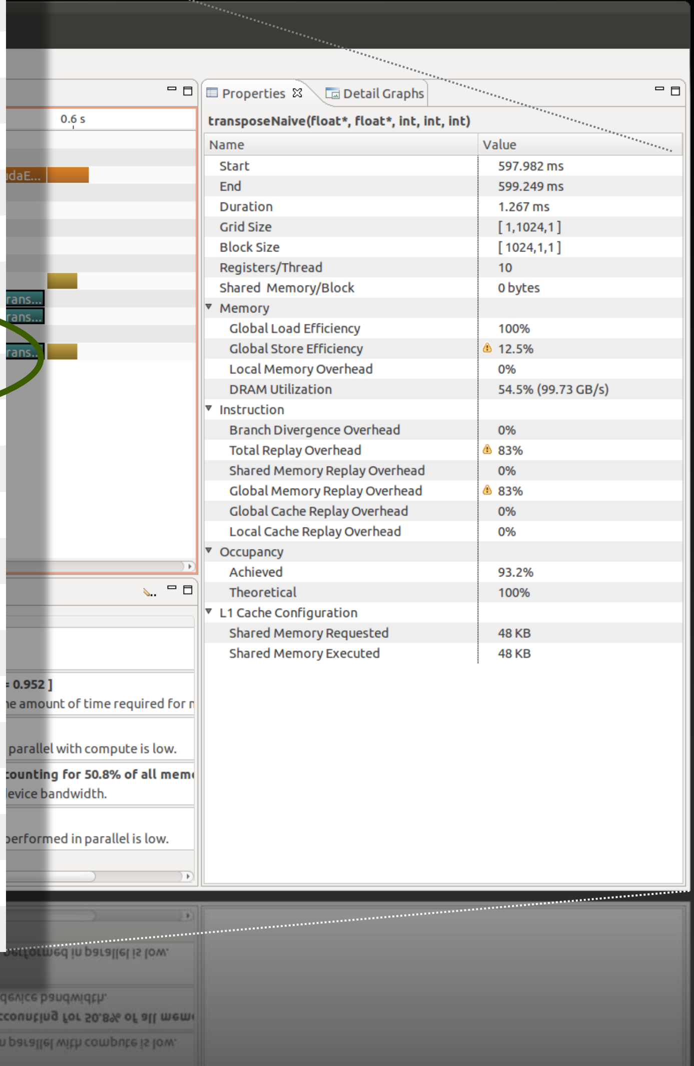


# Application Assessment with NVVP

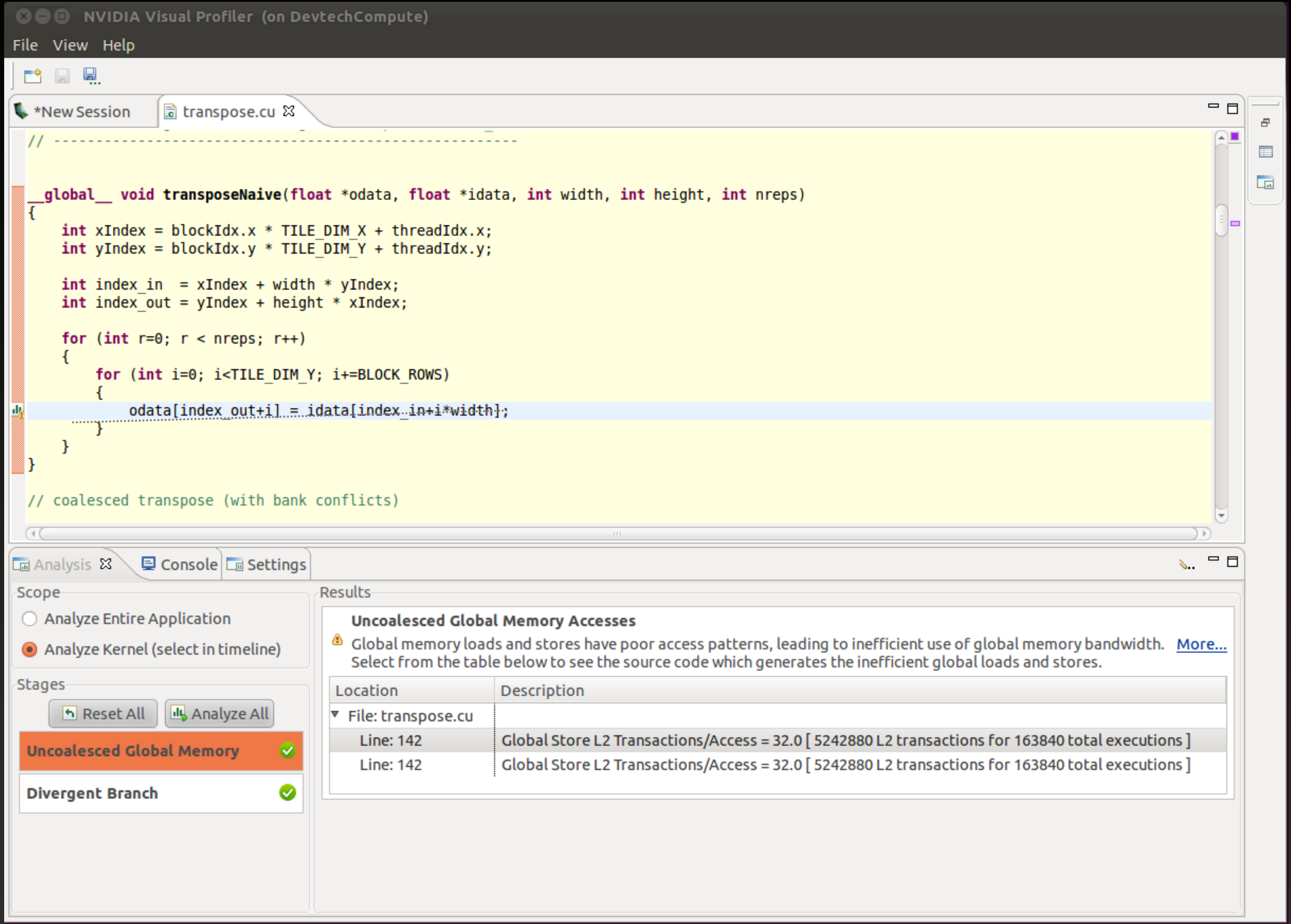


# Application Assessment with NVVP

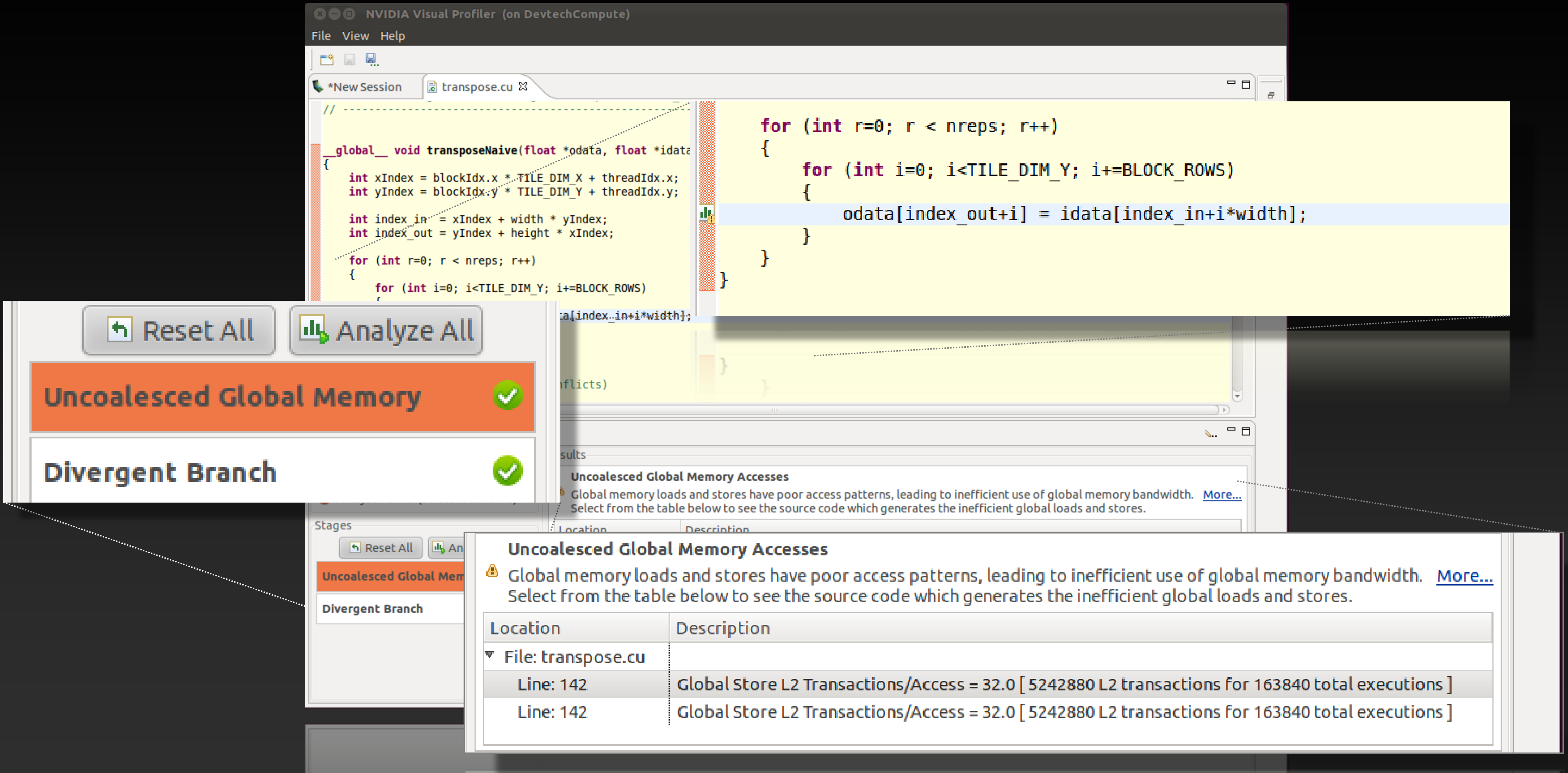
| | |
|---|---|
| Start | 597.982 ms |
| End | 599.249 ms |
| Duration | 1.267 ms |
| Grid Size | [ 1,1024,1 ] |
| Block Size | [ 1024,1,1 ] |
| Registers/Thread | 10 |
| Shared Memory/Block | 0 bytes |
| Memory | |
| Global Load Efficiency | 100% |
| Global Store Efficiency | ⚠ 12.5% |
| Local Memory Overhead | 0% |
| DRAM Utilization | 54.5% (99.73 GB/s) |
| Instruction | |
| Branch Divergence Overhead | 0% |
| Total Replay Overhead | ⚠ 83% |
| Shared Memory Replay Overhead | 0% |
| Global Memory Replay Overhead | ⚠ 83% |
| Global Cache Replay Overhead | 0% |
| Local Cache Replay Overhead | 0% |
| Occupancy | |
| Achieved | 93.2% |
| Theoretical | 100% |



# Source-Level Hot-spot Analysis in NVVP



# Source-Level Hot-spot Analysis in NVVP



# Additional Metrics

| | |
|---|---|
| Start | 588.755 ms |
| End | 588.808 ms |
| Duration | 53.344 μs |
| Grid Size | [ 64,64,1 ] |
| Block Size | [ 16,8,1 ] |
| Registers/Thread | 21 |
| Shared Memory/Block | 1.062 KB |
| Memory | |
| Global Load Efficiency | 100% |
| Global Store Efficiency | 100% |
| Local Memory Overhead | 0% |
| DRAM Utilization | 92.7% (169.74 GB/s) |
| Instruction | |
| Branch Divergence Overhead | 0% |
| Total Replay Overhead | 17.6% |
| Shared Memory Replay Overhead | 0% |
| Global Memory Replay Overhead | 17.6% |
| Global Cache Replay Overhead | 0% |
| Local Cache Replay Overhead | 0% |
| Occupancy | |
| Achieved | 91.3% |
| Theoretical | 100% |

# Alternatives to NVVP: nvprof

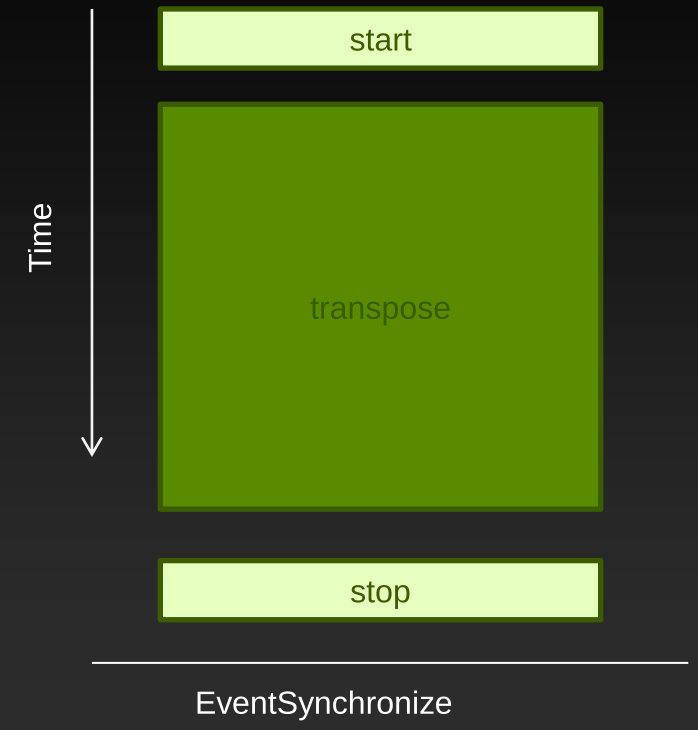
```
%nvprof  --print-gpu-trace ./transpose

Profiling result:
   Start  Duration Grid Size  Block Size  Regs*  Size    Throughput   Name
577.11ms  874.57us         -           -      -  4.19MB  4.80GB/s     [CUDA memcpy HtoD]
598.45ms    1.67ms   (1 1 1)  (1024 1 1)    22      -          -      transposeNaive(float*,
600.12ms    1.67ms   (1 1 1)  (1024 1 1)    22      -          -      transposeNaive(float*,
601.79ms    1.67ms   (1 1 1)  (1024 1 1)    22      -          -      transposeNaive(float*,
```

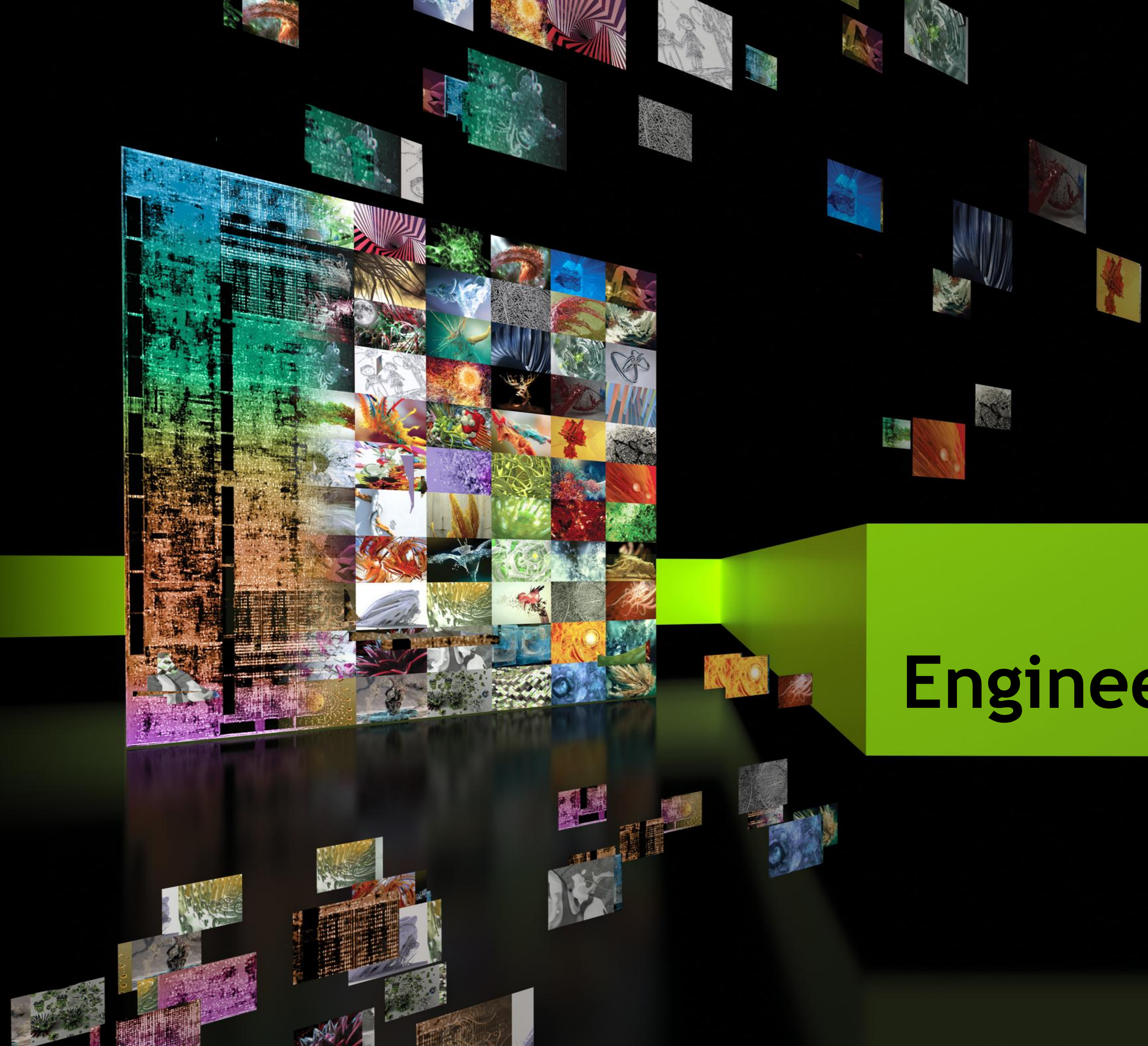
```
nvprof --print-gpu-trace  --aggregate-mode-off --events sm_cta_launched ./transpose

Profiling result:
Device                 Event Name,       Kernel,                          Values
  0                    sm_cta_launched,  transposeNaive(float*, ..), 76 73 72 72 73 74 75 73 73 72 73 73 72 73
```

- Command-Line Profiler
- Access to hardware counters
- List of supported counters: --query-events

# Alternatives to NVVP: Instrumentation

```
cudaEventRecord(start, 0);

transpose<<<grid, threads>>>(..);

cudaEventRecord(stop,0);

cudaEventSynchronize(stop);

cudaEventElapsedTime(&time, start, stop);
```



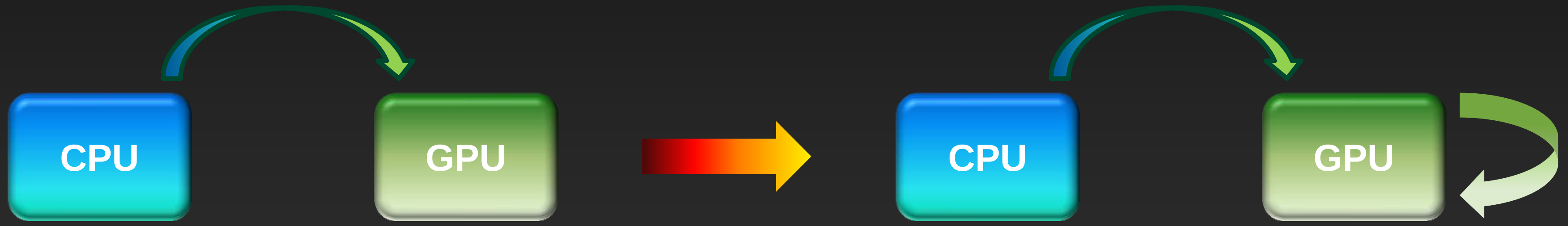


# Engineering

# What is Dynamic Parallelism?

## The ability to launch new grids from the GPU

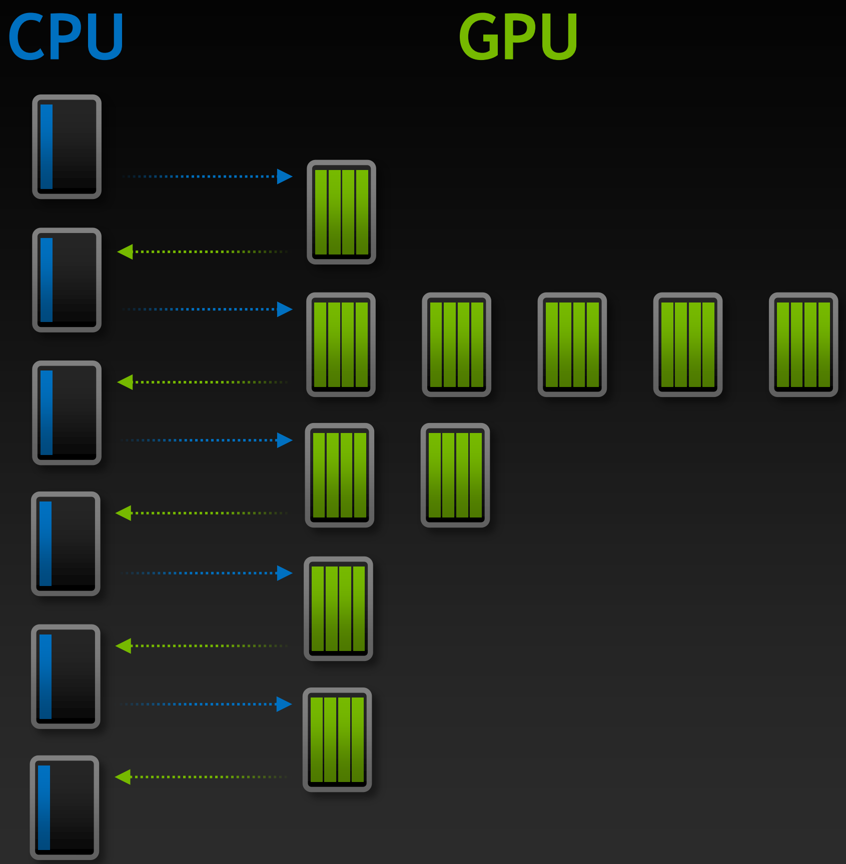
- Dynamically
- Simultaneously
- Independently



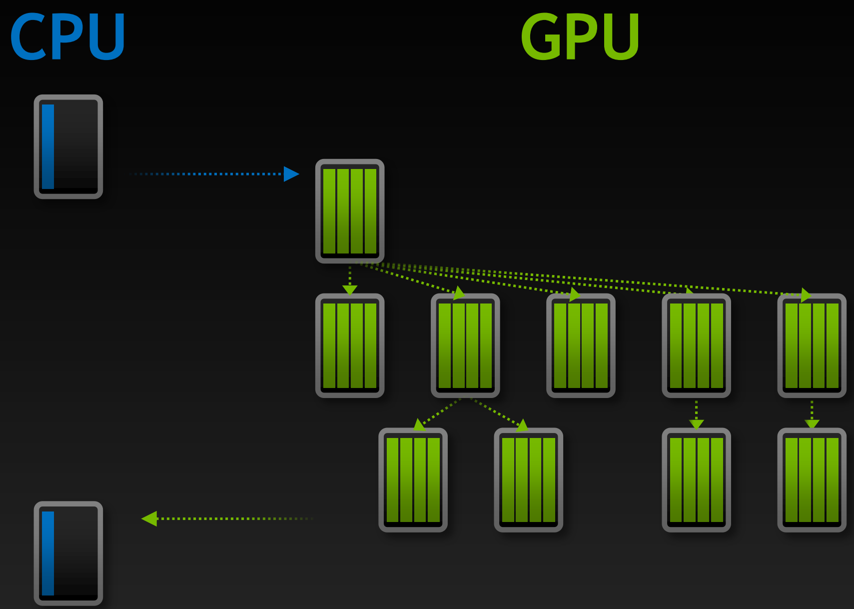

*Fermi: Only CPU can generate GPU work*

*Kepler: GPU can generate work for itself*

# What Does It Mean?

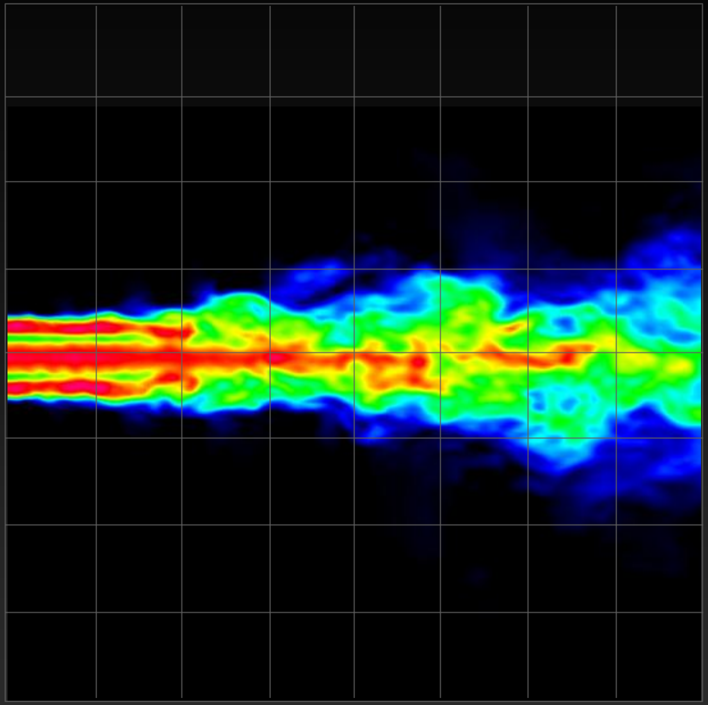



*GPU as Co-Processor*




*Autonomous, Dynamic Parallelism*

# Dynamic Work Generation

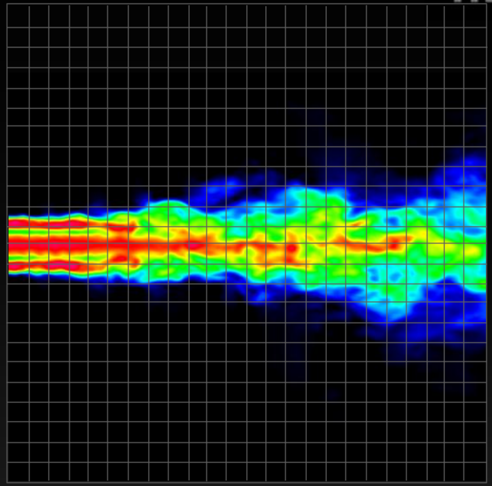


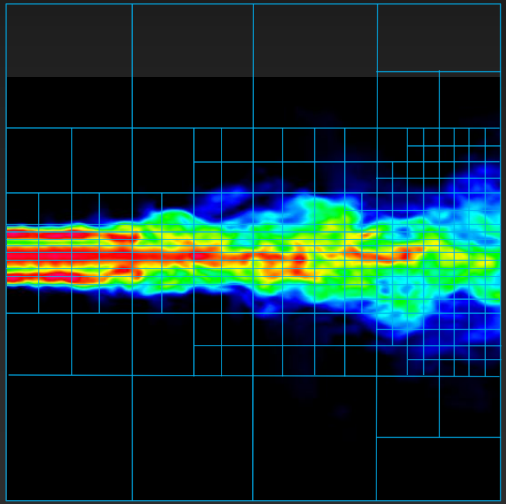
Initial Grid

Statically assign conservative worst-case grid

Dynamically assign performance where accuracy is required

Fixed Grid





Dynamic Grid

# CUDA Dynamic Parallelism

Kernel launches grids

Identical syntax as host

CUDA runtime function in cudadevrt library

Enabled via nvcc flag

-rdc=true

```
__global__ void childKernel()
{
 printf("Hello %d", threadIdx.x);
}
```

```
__global__ void parentKernel()
{
    childKernel<<<1,10>>>();
    cudaDeviceSynchronize();
    printf("World!\n");
}
```

```
int main(int argc, char *argv[])
{
  parentKernel<<<1,1>>>();
  cudaDeviceSynchronize();
  return 0;
}
```

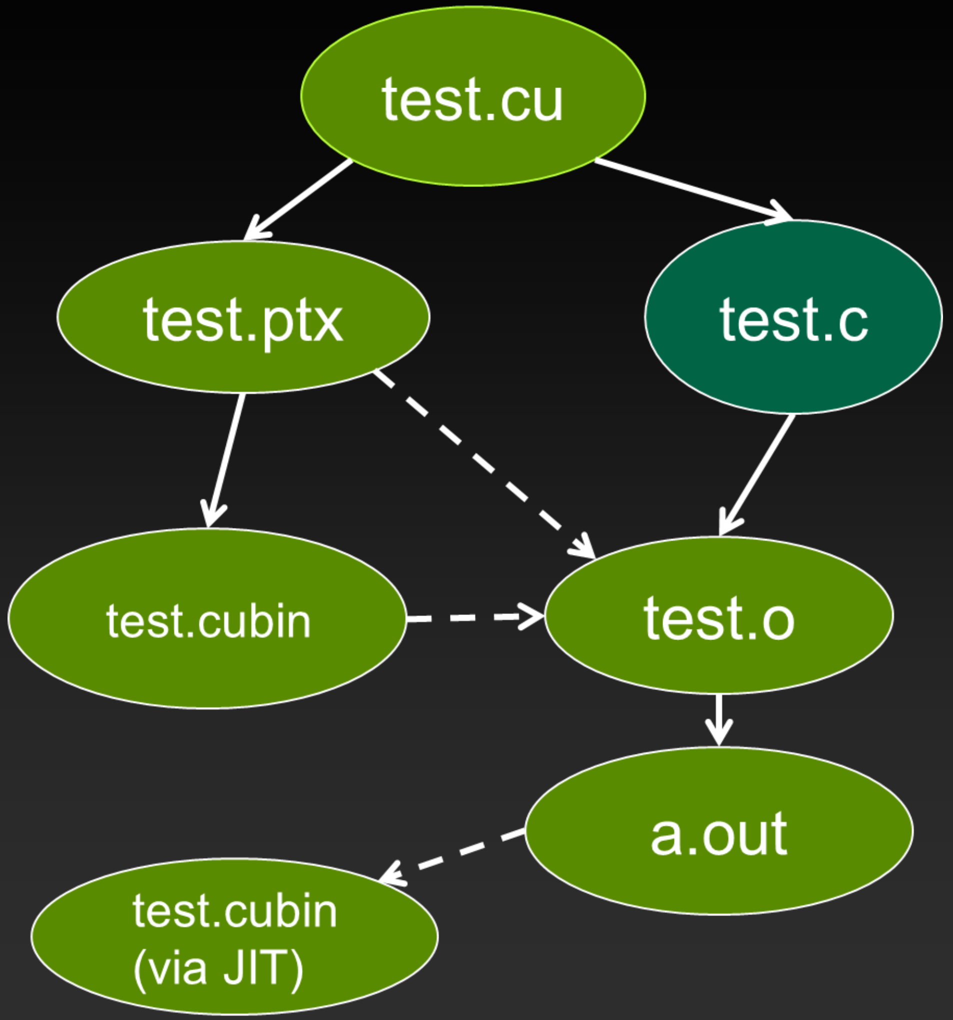
# GPU-Callable Libraries

New in CUDA 5.0

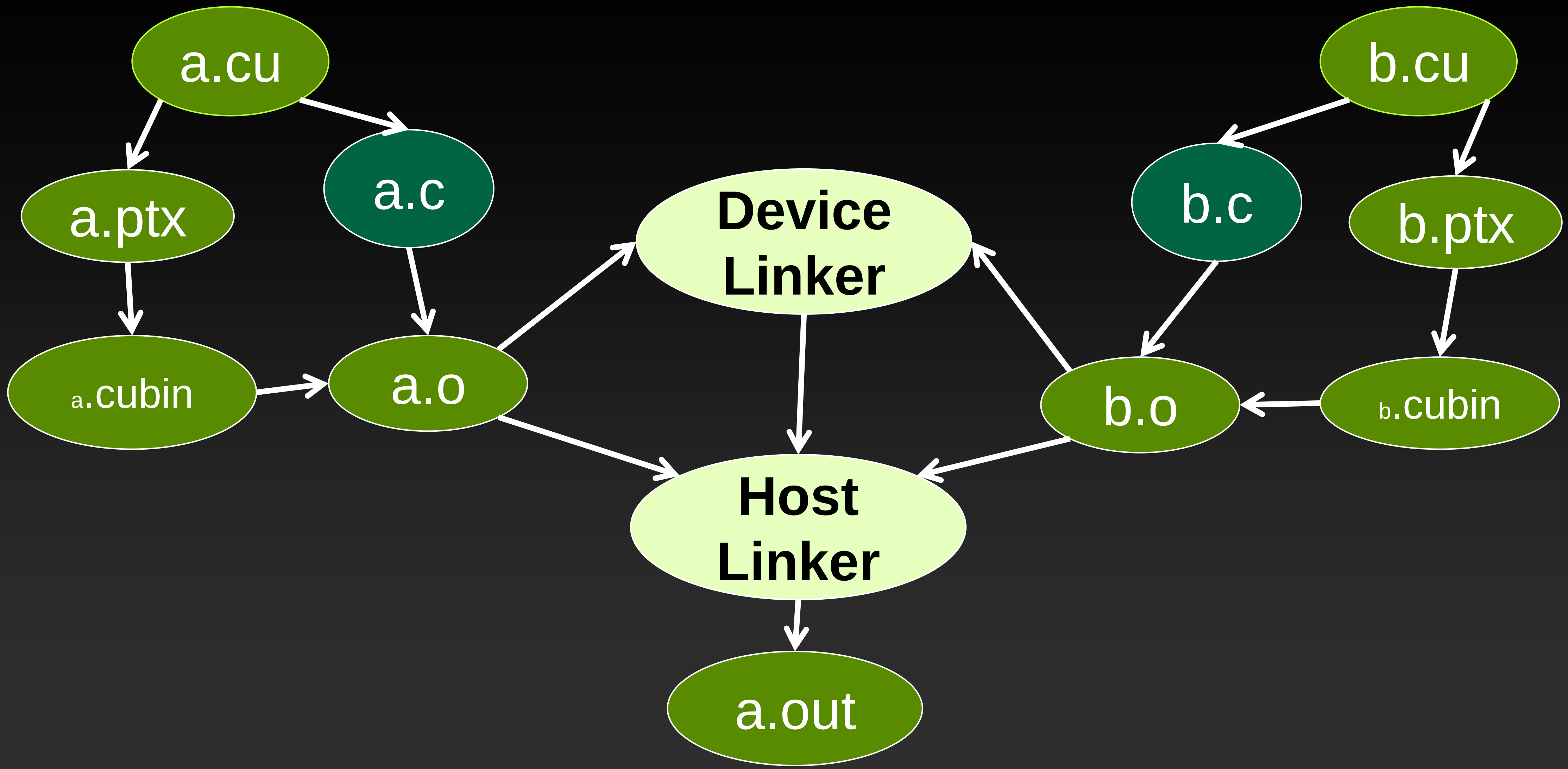
- Call cuBLAS library function from GPU code
- Supported on K20 and K20x only
- Encourages third party libraries

# Compile Trajectory

- Separation of host and device code
- Device code translates into device-specific binary (.cubin) or device independent assembly (.ptx)
- Device code embedded in host object data



# CUDA 5 Introduces Device Code Linker



# Device Linker Invocation

- Introduction of an optional link step for device code

```
nvcc -arch=sm_20 -dc a.cu b.cu
nvcc -arch=sm_20 -dlink a.o b.o -o link.o
g++ a.o b.o link.o -L<path> -lcudart
```

- Link device-runtime library for dynamic parallelism

```
nvcc -arch=sm_35 -dc a.cu b.cu
nvcc -arch=sm_35 -dlink a.o b.o -lcudadevrt -o link.o
g++ a.o b.o link.o -L<path> -lcudadevrt -lcudart
```

- Currently, link occurs at cubin level (PTX not supported)

# Thank you