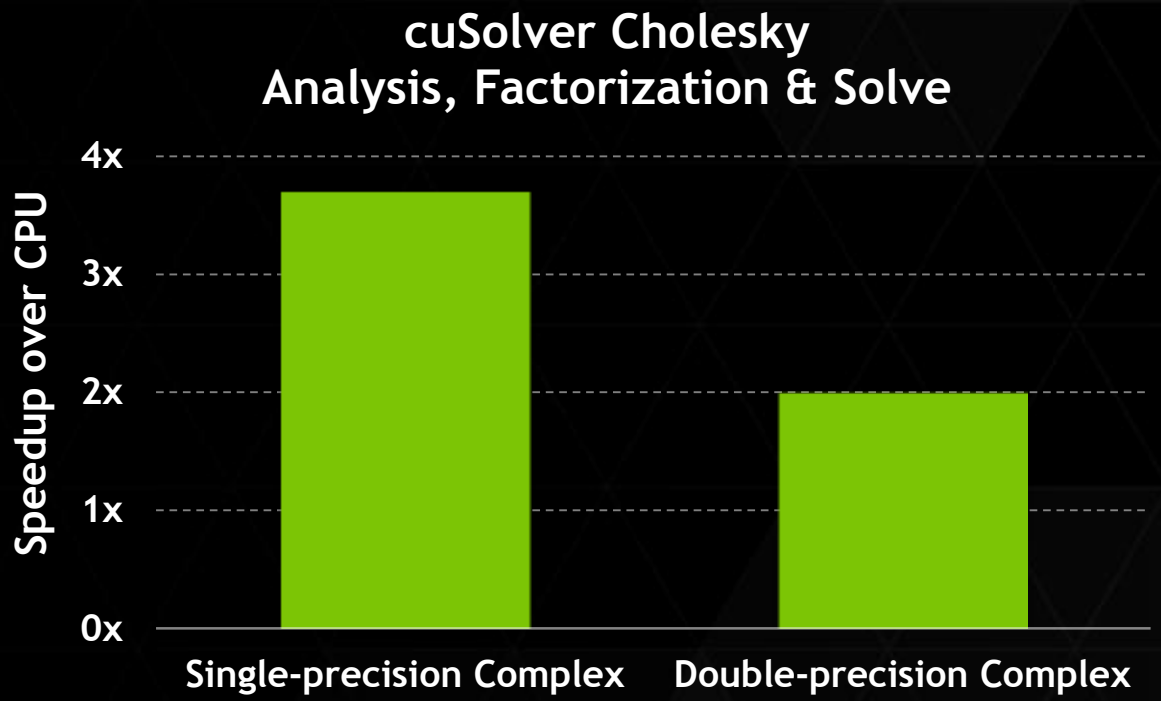
CUDA 7

www.nvidia.com/getcuda

- New cuSOLVER library
  - Accelerates key LAPACK routines, 12x faster direct sparse solvers
- C++11 language features
  - Increases productivity with lambdas, auto, and more
- Runtime Compilation
  - Enables highly optimized kernels to be generated at runtime



cuSolver CUDA 7.0, M=N=4096, K40, 12GB RAM
MKL 11.0.4, 14 core Xeon E5-2697v3 CPU @ 3.6GHz

CUDA 7

1. C++11
2. Thrust Improvements
3. cuSOLVER
4. cuFFT Performance Improvements
5. Runtime Compilation
6. GPU Coredumps
7. New cuda-memcheck tools
8. Per-thread default stream
9. Multi-gpu Multi-Process Server
10. Network installers
11. Platform Support

C++11 SEEING STRONG ADOPTION

NVCC TO SUPPORT C++11 IN HOST AND DEVICE CODE

- C++11 language features enabled, including:
  - Auto
  - Lambda
  - std::initializer_list
  - Variadic Templates
  - Static_asserts
  - Constexpr
  - Rvalue references
  - Range based for loops
  - …
- Not supported:
  - thread_local
  - Standard libraries
    - std::thread,
    - Etc.

Mark Harris' GTC Talk
http://on-demand.gputechconf.com/gtc/2015/video/S5820.html

Parallel Forall Blog Post:
http://nvda.ly/Kty6M

C++11 “auto” and Thrust

- Naming complex Thrust iterator types can be troublesome:

```
typedef typename device_vector<float>::iterator FloatIterator;
typedef typename tuple<FloatIterator,
                       FloatIterator,
                       FloatIterator> FloatIteratorTuple;
typedef typename zip_iterator<FloatIteratorTuple> Float3Iterator;
Float3Iterator first =
    make_zip_iterator(make_tuple(A0.begin(), A1.begin(), A2.begin()));
```

- C++11 auto makes it easy! Variable types automatically inferred:

```
auto first = make_zip_iterator(make_tuple(A0.begin(), A1.begin(), A2.begin()));
```

CUDA 7 Thrust Improvements

- Faster algorithms
  - **`sort`**, **`scan`**, **`reduce`**, **`reduce_by_key`**, **`merge`**
- API Support for CUDA stream argument (concurrency between threads)
- Support for device-side API
  - Can call Thrust API from kernels for:
    - sequential execution
    - parallel launch via dynamic parallelism

New: Device-Side Thrust

- Call Thrust algorithms from CUDA device code

```
__global__
void xyzw_frequency_thrust_device(int *count, char *text, int n)
{
  const char letters[] { 'x','y','z','w' };

  *count = thrust::count_if(thrust::device, text, text+n, [=](char c) {
    for (const auto x : letters)
      if (c == x) return true;
    return false;
  });
}
```

Device Execution

Device Lambda

  - Device execution uses Dynamic Parallelism kernel launch on supporting devices
  - Can also use `thrust::cuda::par` execution policy

New: Device-Side Thrust

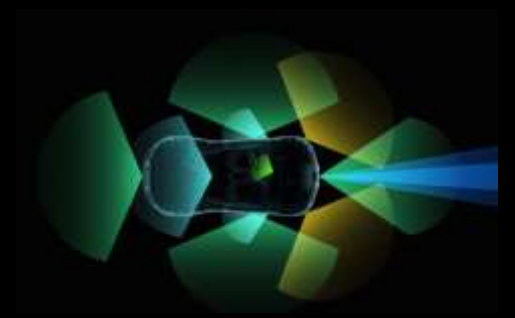
- Call Thrust algorithms from CUDA device code

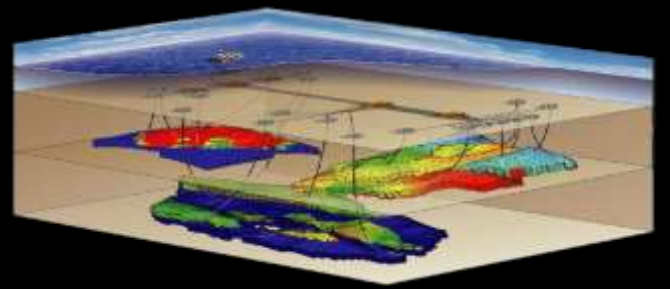
```
__global__
void xyzw_frequency_thrust_device(int *count, char *text, int n)
{
  const char letters[] { 'x','y','z','w' };

  *count = thrust::count_if(thrust::seq, text, text+n, [=](char c) {
    for (const auto x : letters)
      if (c == x) return true;
    return false;
  });
}
```

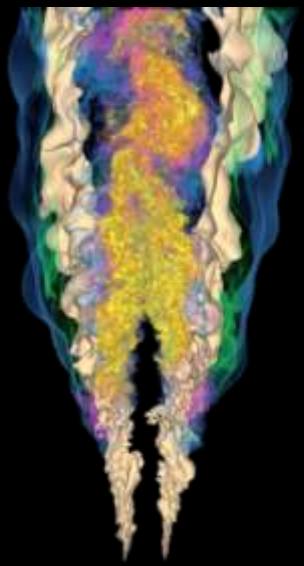
Sequential Execution
Within each CUDA thread

cuSOLVER

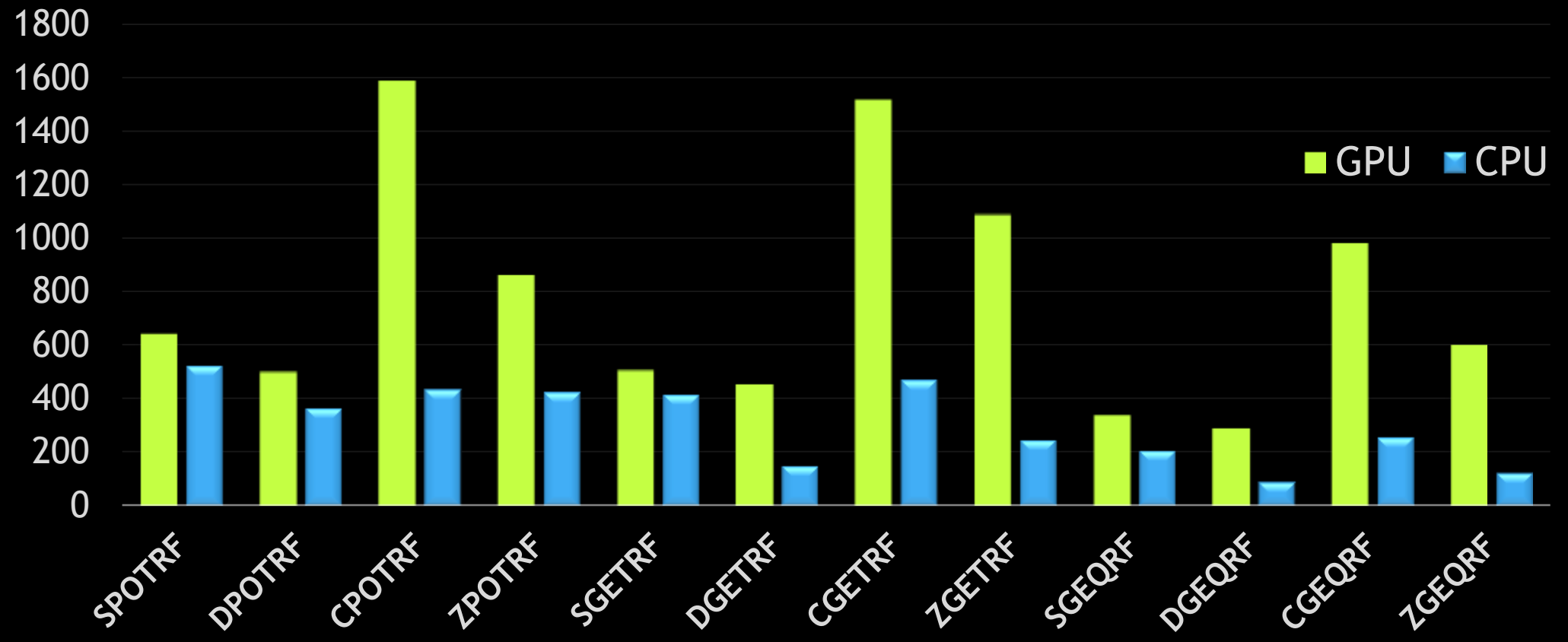
- cusolverDN
  - Dense Cholesky, LU, SVD, (batched) QR
  - Optimization, Computer vision, CFD
- cusolverSP
  - Sparse direct solvers & Eigensolvers
  - Newton's method, Oil & Gas Well Models
- cusolverRF
  - Sparse refactorization solver
  - Chemistry, ODEs, Combustion, Circuit simulation





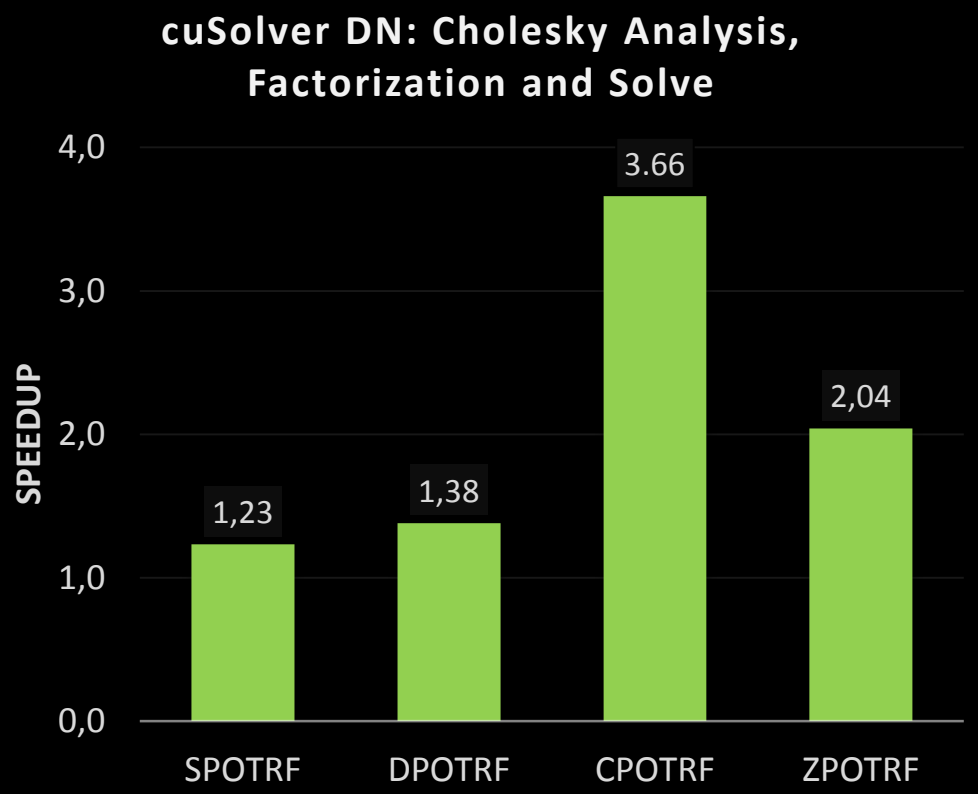


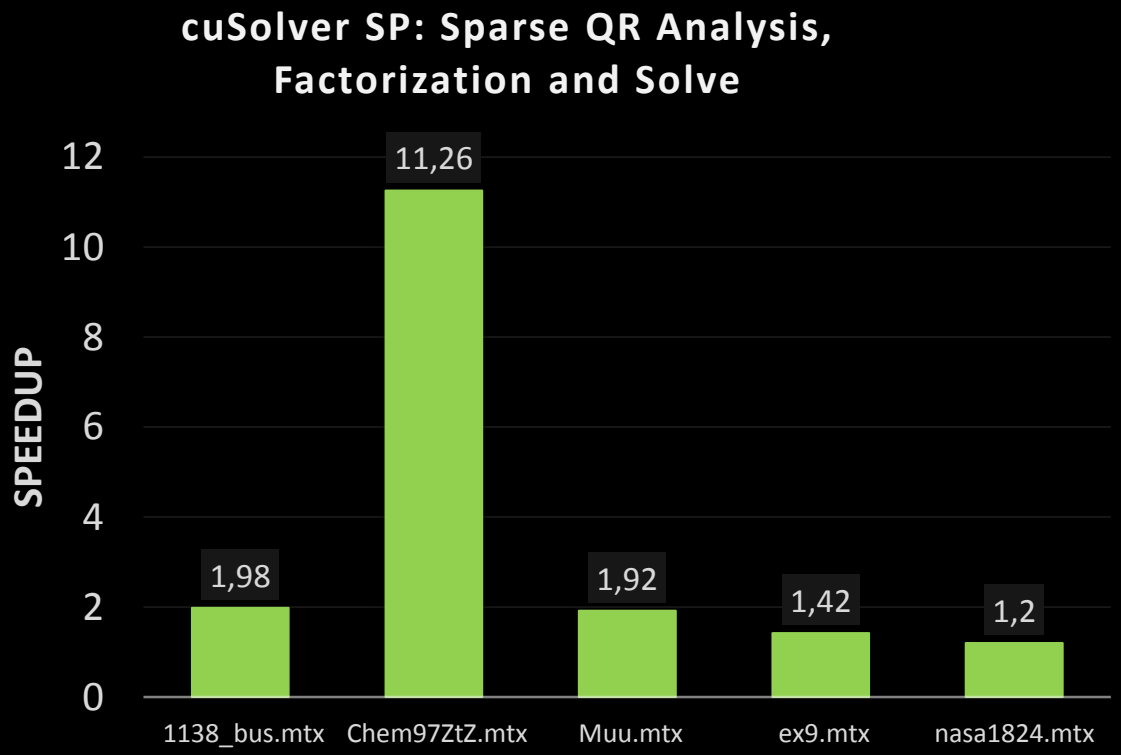
cuSOLVER Dense Gflops vs MKL



GPU:K40c M=N=4096
CPU: Intel(R) Xeon(TM) E5-2697 v3 CPU @ 3.60GHz, 14 cores
MKL v11.04

cuSOLVER Speedup

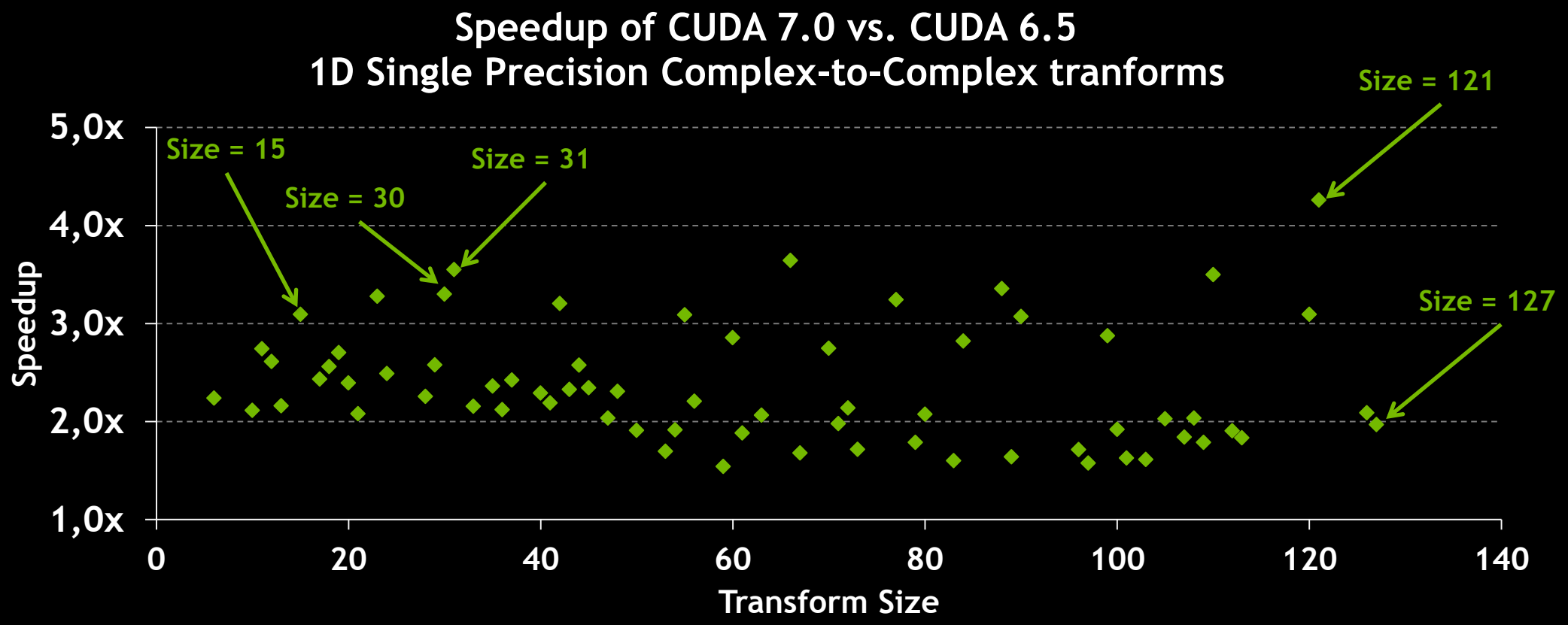




GPU:K40c M=N=4096
CPU: Intel(R) Xeon(TM) E5-2697v3 CPU @ 3.60GHz, 14 cores
MKL v11.04 for Dense Cholesky, Nvidia csr-QR implementation for CPU and GPU

cuFFT Performance improvements

2x-3x speedup for sizes that are composite powers of 2, 3, 5, 7 & small primes

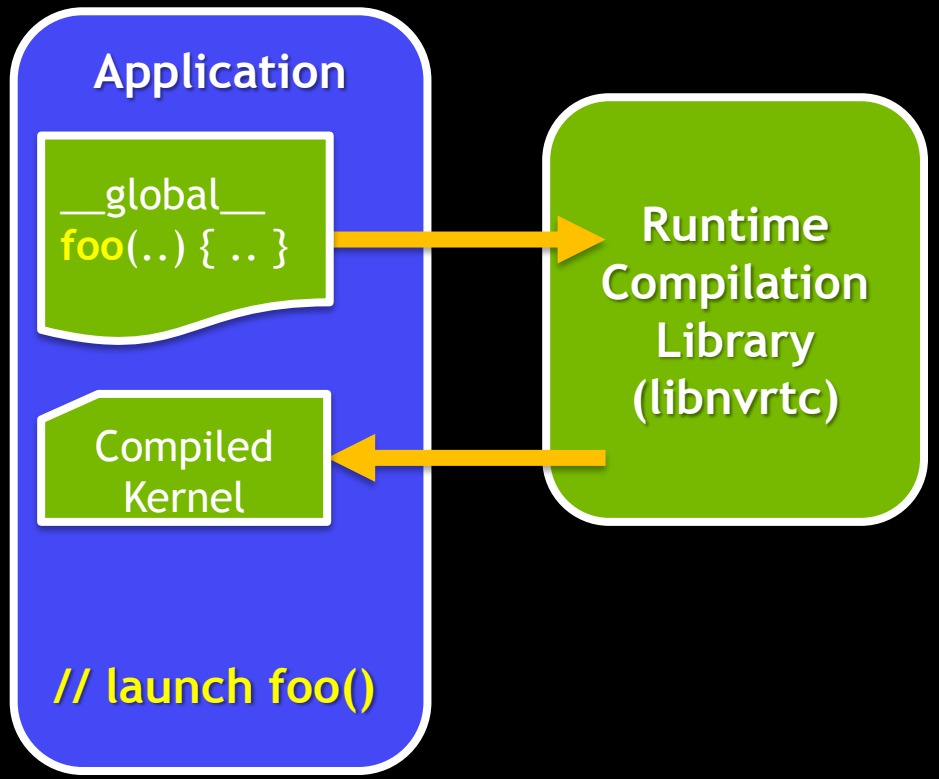


• cuFFT 6.5 and 7.0 on K20c, ECC ON, Batched transforms on 32M total elements, input and output data on device

CUDA Runtime Compilation

Preview feature in CUDA 7.0 (API may change; no tools support)

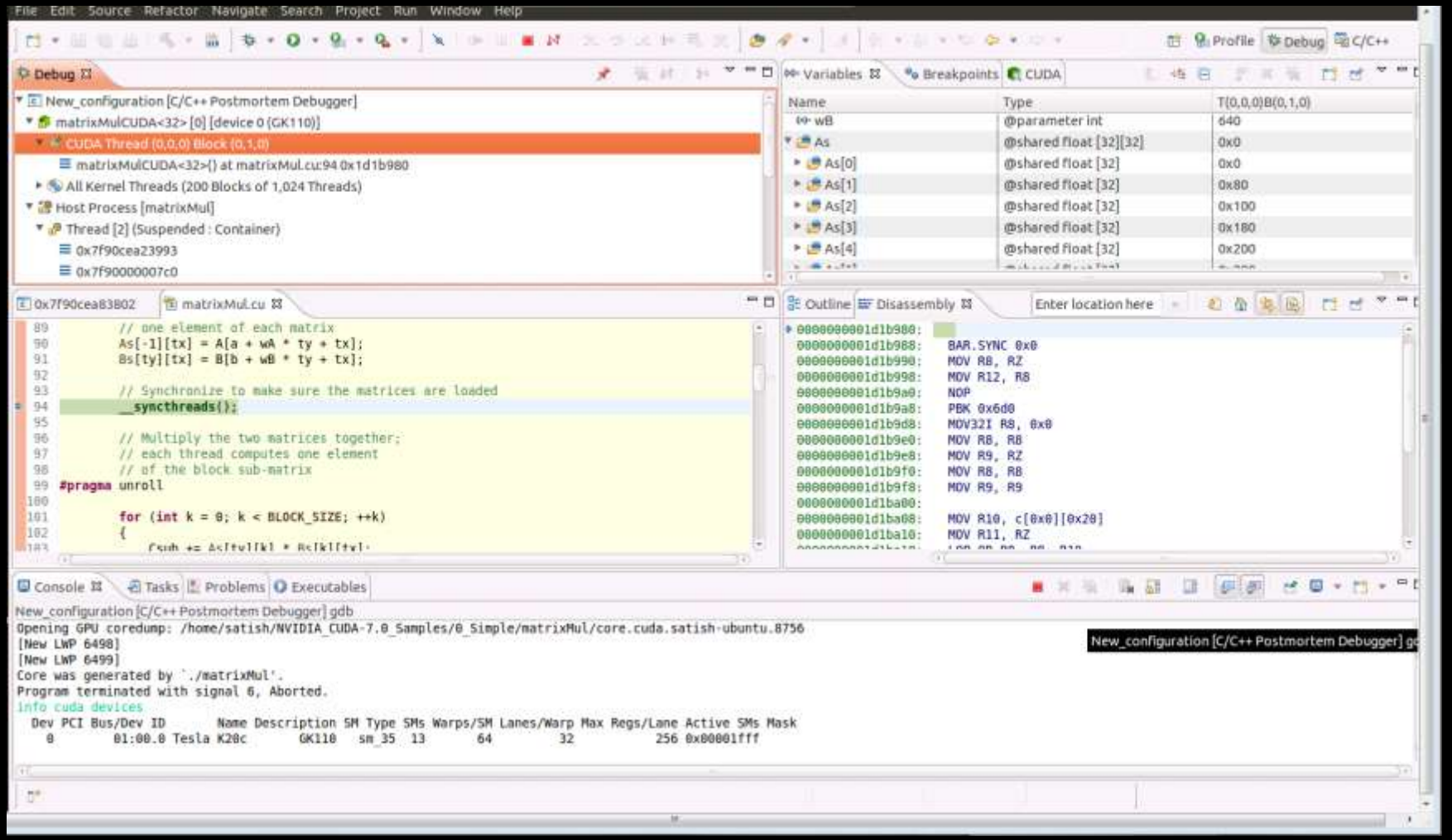
- Compile CUDA kernel source at run time
  - Compiled kernels can be cached on disk
- *Run-time* C++ template specialization
  - Before: Compile templates multiple times—once for each datatype that *might* be needed by any user
  - Now: only compile for datatypes a specific user needs
  - Reduces compile time and compiled code size
  - No compromise on performance
- Simplifies deployment of DSLs that target CUDA C instead of NVVM IR



GPU Core Dump - Cluster Debugging

- No need to bring down a cluster node
- Opt-in feature supported on multiple platforms - Linux, MAC OSX, Windows(TCC) and L4T
- Generates both CPU and GPU core dumps
- (cuda-gdb) target core core.cpu core.cuda.host.pid

Load Core Dumps into Nsight Eclipse Edition



New CUDA-MEMCHECK tool: INITCHECK

- Detect un-initialized (global) device memory usage
  - `cuda-memcheck --tool initcheck`

```
__global__ void kernel(int* a)
{
    (*a)++;
}

int main(void)
{
    int *d_ptr;
    cudaMalloc((void**)&d_ptr,
               sizeof(d_ptr));

    kernel<<<1,1>>>(d_ptr);
    cudaDeviceSynchronize();
    return 0;
}
```

```
$ cuda-memcheck --tool initcheck /tmp/test
========= CUDA-MEMCHECK
========= Uninitialized __global__ memory read of size 4
=========     at 0x000ab320 in kernel(int*)
=========     by thread (0,0,0) in block (0,0,0)
=========     Address 0x5047a0000
=========     Saved host backtrace up to driver entry point
=========     Host Frame:<13 frames were hidden>
```

New CUDA-MEMCHECK tool: SYNCCHECK

- Detect barriers in use within divergent code
  - `cuda-memcheck --tool synccheck`

```
__device__ int x;
__global__ void kernel(void)
{
    if (threadIdx.x) {
        x = 1;
        __syncthreads();
    }
    else {
        x = 0;
        __syncthreads();
    }
}

int main(void)
{
    kernel<<<1, 2>>>();
    cudaDeviceSynchronize();
    return 0;
}
```

```
$ cuda-memcheck --tool synccheck /tmp/test
========= CUDA-MEMCHECK
========= Barrier error detected. Encountered barrier with
divergent threads in block
=========     at 0x00000058 in kernel(void)
=========     by thread (1,0,0) in block (0,0,0)
=========     Device Frame:kernel(void) (kernel(void) : 0x58)
```

PER-THREAD DEFAULT STREAM

```
k_1 <<< 1, 1, 0, my_stream >>>();
k_2 <<< 1, 1, 0, 0         >>>();
```

EXISTING BEHAVIOR

- If compiled with:
  - No options *OR*
    --default-stream null *OR*
    --default-stream legacy
- k2 synchronizes with k1
- k2 synchronizes with all streams in other threads

NEW OPT-IN BEHAVIOR

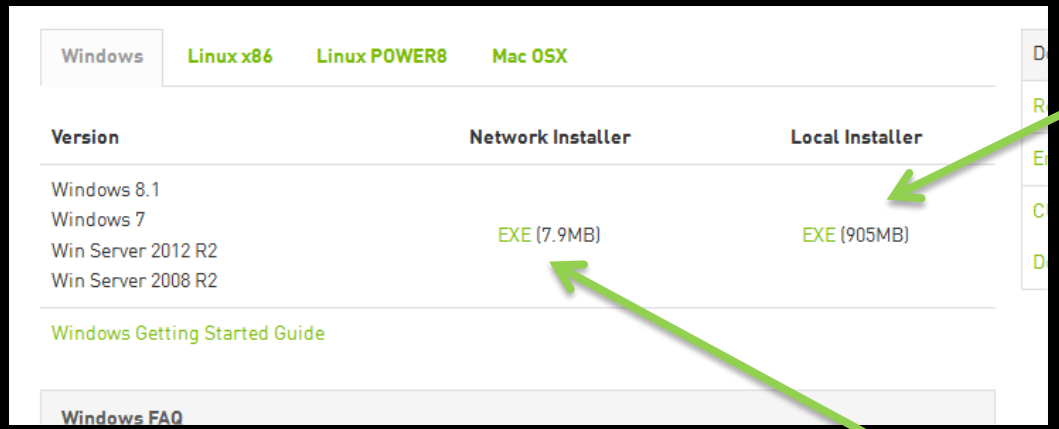
- If compiled with:
  - --default-stream per-thread
- k2 executes asynchronously from k1
- k2 executes asynchronously from all streams in other threads (except legacy stream 0)

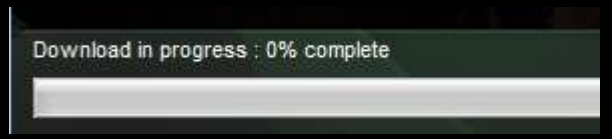
OTHER SOFTWARE FEATURES

See release notes and documentation for more details

- MPS Server now supports more than one gpu per node
- LLVM standard C++ library (libc++) on Mac OSX
- Separate licenses no longer needed to use cuBLAS-XT or cuFFT-XT
- cuFFT-XT multi-gpu execution support increased to 4 GPUs
- New 3D/4D Euclidean norm & 3D Euclidean reciprocal norm functions
- Double precision reciprocal, rcp(), significantly optimized

Network Installers





- Local installer
  - All components embedded within installer (toolkit, driver, samples)
  - Large: ~1GB
- Network installer (NEW in 7.0)
  - Only components chosen for installation are downloaded
  - Small: <10MB
  - Windows and Mac

PLATFORM SUPPORT

- Linux
  - RHEL & CentOS 6, 7
  - Fedora 21 Workstation [Coming Soon]
  - SLES 11 SP3, 12
  - OpenSUSE 13.1, 13.2
  - Ubuntu 12.04 & 14.04 LTS, 14.10
  - SteamOS 1.0-beta
- Windows
  - 7, 8.1, Server 2008 R2, 2012 R2
  - Visual Studio 2010, 2012, 2013 [CE]
- Mac OSX 10.9, 10.10

- Alternative Linux host compilers
  - Intel `icc` 15.0.0
  - PGI `pgc++` 14.9(+)
  - IBM xlc/xlC 13.1.1 (only on POWER8)
- CUDA 7.0 drops support for:
  - RHEL 5, CentOS 5
  - 32-bit Linux Systems
  - SM 1.0 – 1.3 GPU Architectures
  - `gcc` on Mac OSX (use clang instead)