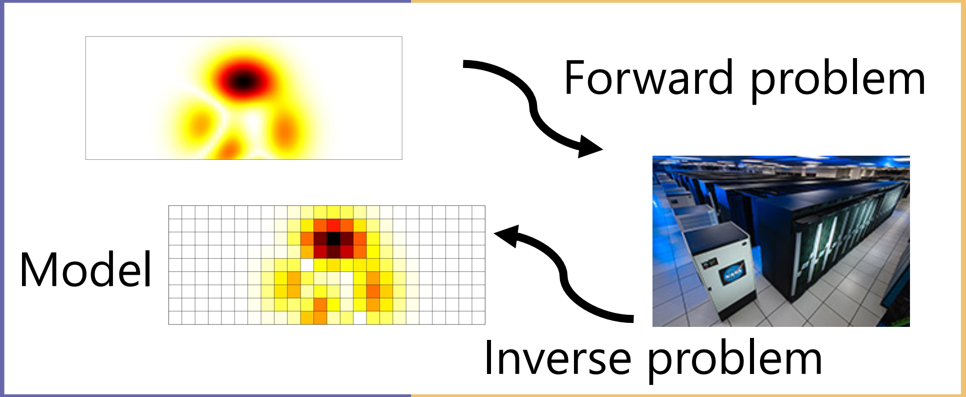
ADDING GPU SUPPORT TO THE MARKOV CHAIN MONTE CARLO CODE CATMIP

Sarah Minson + Gabriele Jost

WHAT? AND WHY?

- In geophysics we have a lot of under-determined inverse problems
  - More that one model fits the data
  - e.g., there are lots of possible fault slip models that produce the same surface deformation
- What if instead of choosing one model, we chose all the models?



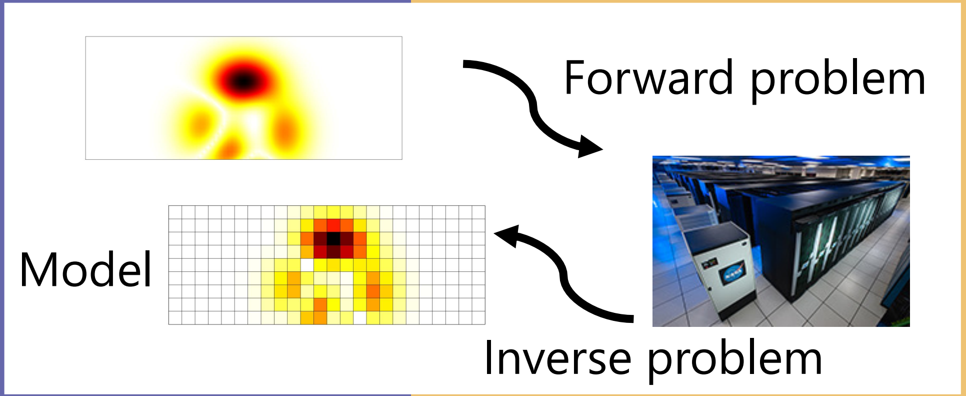
BAYESIAN MODELING

Posterior probability:
Relative plausibility of all potential values of model parameters

- $p(\theta|d) \sim p(d|\theta) \times p(\theta)$

Data likelihood:
How well a set of model parameter values predict the data – we often use normal distribution (i.e., L2 norm)

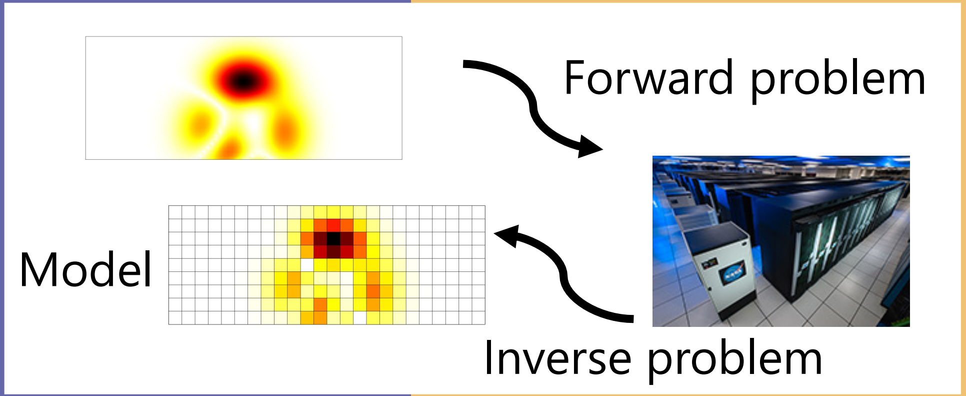
Prior probability:
Our *a priori* beliefs about which model parameter values are likely (e.g., for slip models, don't let the fault slip backwards)



EXAMPLE: EARTHQUAKE SLIP MODEL

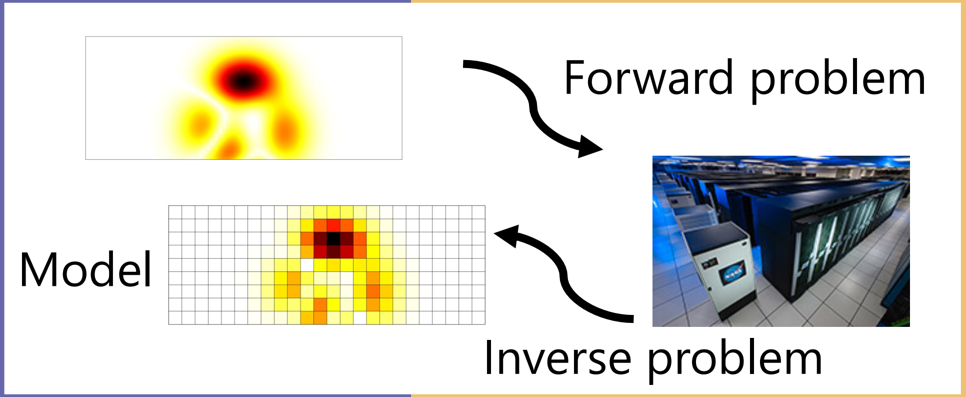
- Build a rupture by embedding a fault (discretized mesh) inside the Earth (a prescribed elastic structure)
- Solve for slip amplitude on each discretized fault patch
- Elastic materials deform linearly → simple linear problem
    - Surface observations
    - = precomputed Earth response
    - X slip on fault patches

Linear but *under-determined* problem. Need to use Bayesian inference to find all plausible slip values.



HOW TO MAKE A BAYESIAN MODEL? MONTE CARLO SAMPLING

- p(θ|d) ~ p(d|θ) x p(θ)
- Goal: Draw a bunch of random distributed according to p(θ|d)
- The statistics of your samples are the statistics of the model space
  - The mean of the samples is the mean of the models that fit the data
  - The median of the samples is the median of the models that fit the data
  - The covariance of the samples is the covariance of the model parameters
  - Sort the samples, toss the bottom 2.5% and top 2.5% of samples, and you have the 95% confidence bound on your model parameters



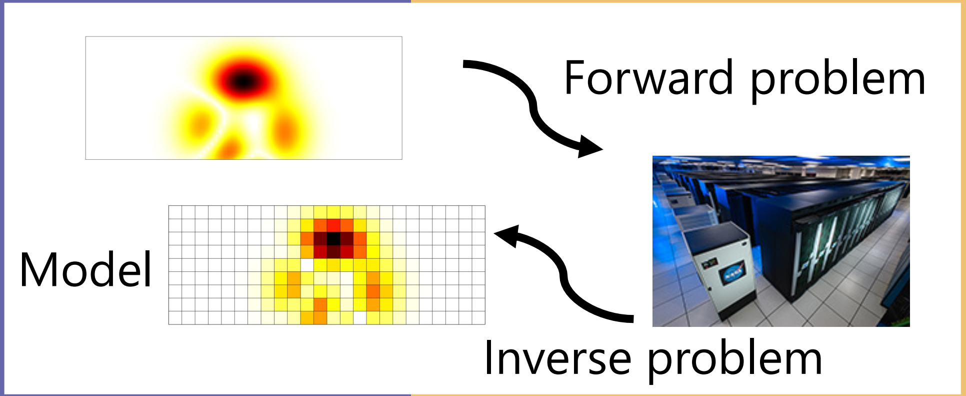
BAYESIAN MODELING

Posterior probability:
Relative plausibility of all potential slip values

- $p(\theta|d) \sim p(d|\theta) \times p(\theta)$

Data likelihood:
How well a set of slip values predict the deformation of the Earth's surface (pre-computed Earth response X slip values)

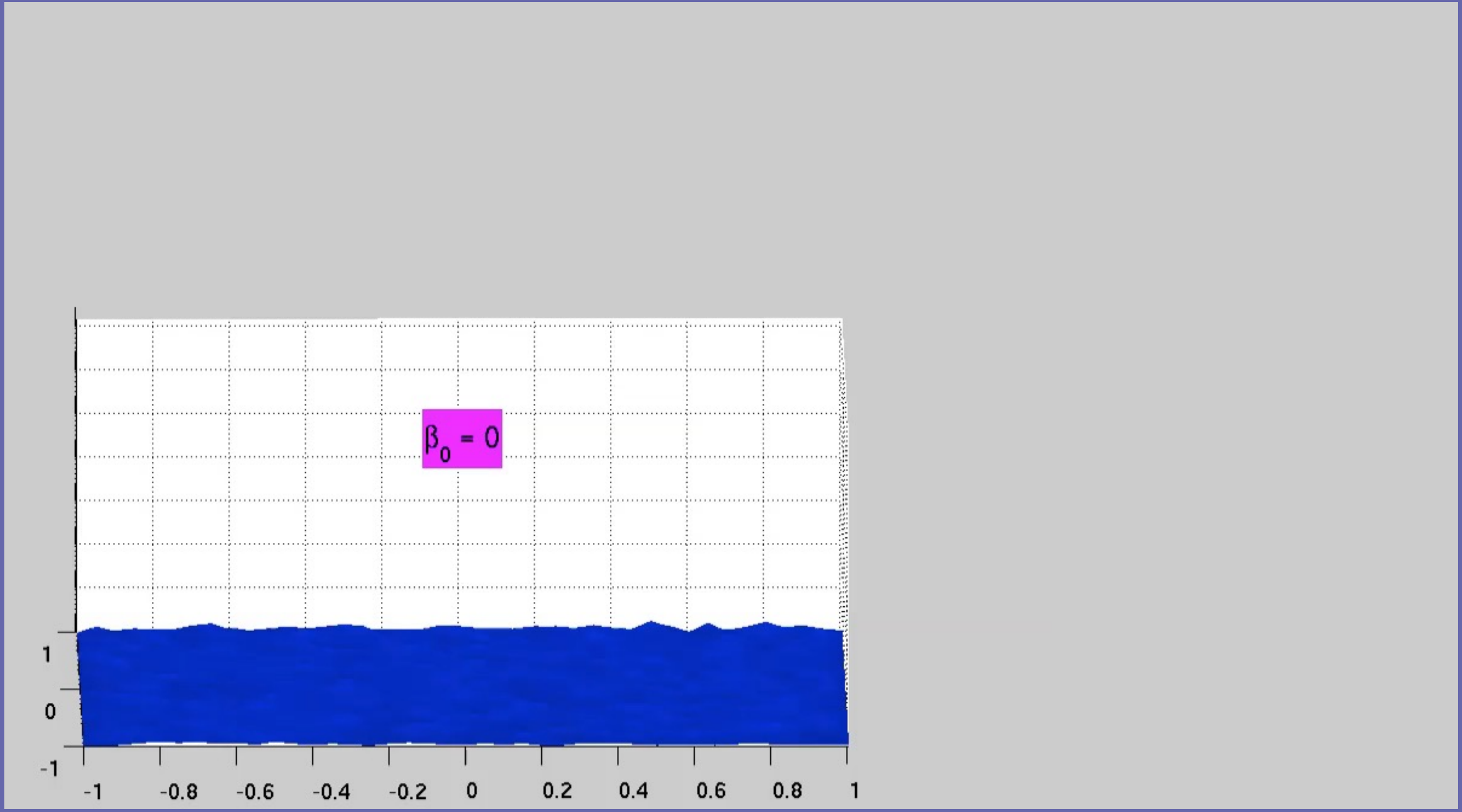
Prior probability:
The fault should not slip backwards



CATMIP

Cascading Adaptive Transitional Metropolis In Parallel (CATMIP)

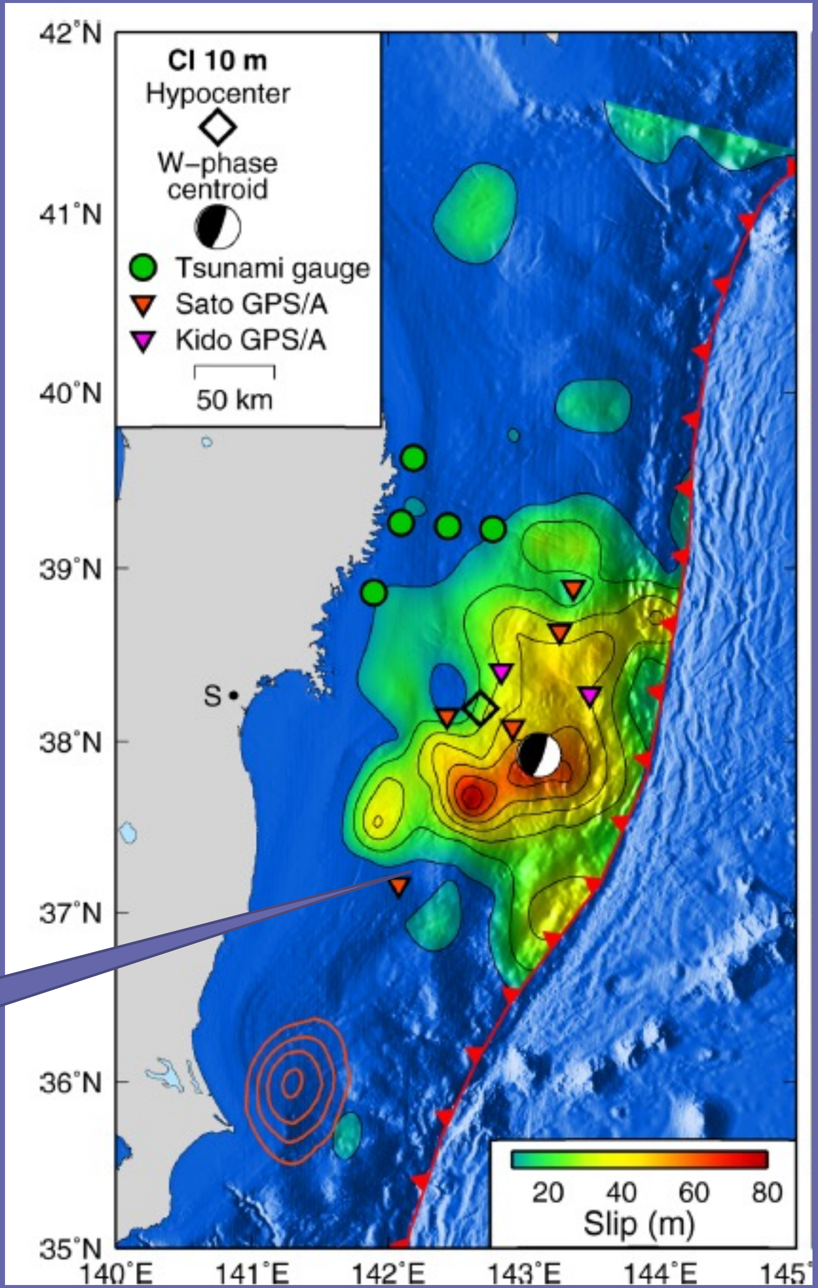
- Parallelized version of TMCMC (Ching and Chen, 2007)
- Includes assorted efficiencies:
  - Multiple parallel Markov chains
    - Embarrassingly parallel
  - Optimized proposal PDF
  - Resampling (killing chains in low probability areas and duplicating chains in high probability areas)
  - **Transitioning** (Beck and Au, 2002)



CATMIP IS A GENERIC SOLVER

- I use it for finite fault earthquake rupture modeling
  - Static and kinematic
- Other people have used it for:
  - Mineral composition of Mars sand dunes
  - History of ocean salinity
  - Earthquake location
  - Block modeling

2011 M9 Tohoku-oki, Japan earthquake
(Minson et al., *GJI*, 2014)



Mean of posterior slip
- 866 free parameters
- ~60 billion MCMC samples

HOW IS CATMIP WRITTEN?

- It uses a **divide-and-conquer** farm:
    - Master-Worker model
    - Many Markov chains are run in parallel with the global sampling parameters periodically optimized by the master process
- Code structure:
    - **libcatmip:** Executes MCMC sampling density proportional to probability returned by functions defined by user (globally maintained, not to be edited by users)
    - **User model library:**
        - Performs I/O and evaluates probability associated with candidate samples generated by libcatmip
        - Model specific with each application problem requiring its own specific instance

CATMIP IMPLEMENTATION DETAILS

- Implemented in C
- Employs MPI for parallelization:
  - Master-Worker Model
  - Master assigns Markov chains to workers
  - Master does not participate in the computation
- Employs BLAS for linear algebra operations and GSL for random number generation
- Strategy for adding GPU support to CATMIP:
  - **Step 1: CPU optimization**
    - Convert the Metropolis Sampling of the Linear Model to employ level 3 rather than level 2 BLAS routines
  - **Step 2: GPU port**
    - Replace the calls to GSL/BLAS calls with calls to CUBLAS and CURAND library calls if possible
    - Write CUDA kernels for the remaining time-consuming loops
  - Compilation:
    - Intel icc/icpc, Intel MKL , cuda/11.0, Nvidia Curand and Cublas

Metropolis Sampling of the Linear Model

- **Original workflow**: Employ level 2 BLAS routines

```
FOR j=1, 2, …, J
FOR k=1,2,…,K
    GENERATE random u from U(0,1)
    GENERATE random q from N(0,C)
    SET y = x + q
    SET llk_y = LLK_POST(y,β)
    SET α = min[exp(llky$_1$-llkx$_1$),1]
    (subscript 1 denotes first element
    of llk_x and llk_y vectors)
    IF u ≤ α
    SET x = y, llk_x = llk_y
    END IF
END FOR
END FOR
```

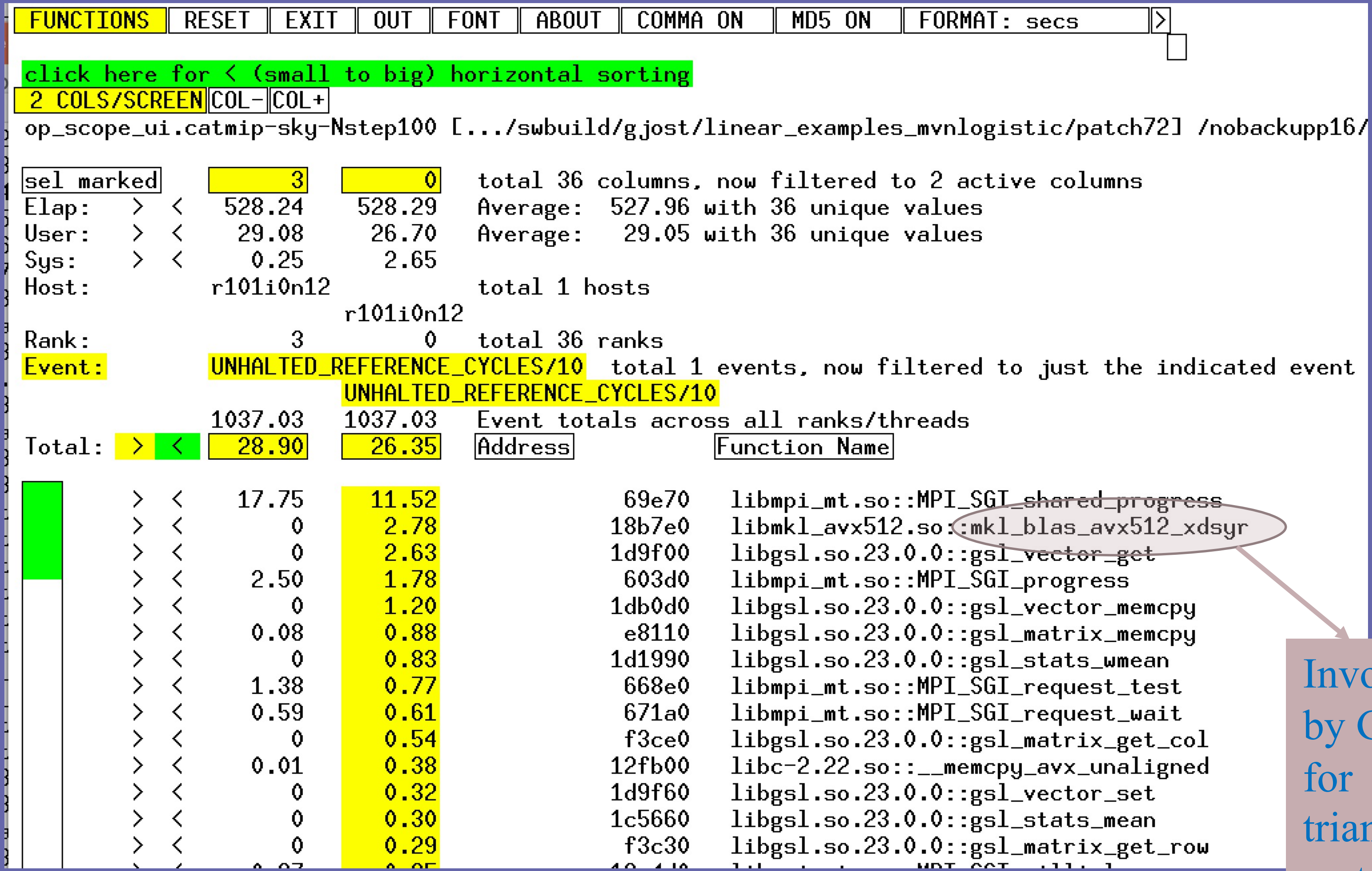
- **Optimized CPU workflow**: Employ level 3 BLAS in Metropolis sampling

  -Evaluate all Markov chains simultaneously

```
FOR k=1,2,…,K
    GENERATE random u from U(0,1)
    GENERATE random q from N(0,C)
    SET y = x + q
    SET llk_y = LLK_POST(y,β)
    SET α = min[exp(llky$_1$-llkx$_1$),1]
    (subscript 1 denotes first element
    of llk_x and llk_y vectors)
    IF u ≤ α
    SET x = y, llk_x = llk_y
    END IF
END FOR
```

- All level 2 BLAS calls are replaced with level 3 BLAS calls
- One loop level is removed

CPU Profile of Optimized Metropolis Sampling



FUNCTIONS | RESET | EXIT | OUT | FONT | ABOUT | COMMA ON | MD5 ON | FORMAT: secs

```
click here for < (small to big) horizontal sorting
 2 COLS/SCREEN COL- COL+
op_scope_ui.catmip-sky-Nstep100 [.../swbuild/gjost/linear_examples_mvnlogistic/patch72] /nobackupp16/

sel marked          3        0   total 36 columns, now filtered to 2 active columns
Elap:     > <  528.24   528.29   Average:  527.96 with 36 unique values
User:     > <   29.08    26.70   Average:   29.05 with 36 unique values
Sys:      > <    0.25     2.65
Host:         r101i0n12          total 1 hosts
                       r101i0n12
Rank:               3        0   total 36 ranks
Event:        UNHALTED_REFERENCE_CYCLES/10  total 1 events, now filtered to just the indicated event
                       UNHALTED_REFERENCE_CYCLES/10
              1037.03  1037.03   Event totals across all ranks/threads
Total:    > <   28.90    26.35   Address          Function Name

          > <   17.75    11.52            69e70   libmpi_mt.so::MPI_SGI_shared_progress
          > <       0     2.78           18b7e0   libmkl_avx512.so::mkl_blas_avx512_xdsyr
          > <       0     2.63           1d9f00   libgsl.so.23.0.0::gsl_vector_get
          > <    2.50     1.78           603d0    libmpi_mt.so::MPI_SGI_progress
          > <       0     1.20           1db0d0   libgsl.so.23.0.0::gsl_vector_memcpy
          > <    0.08     0.88            e8110   libgsl.so.23.0.0::gsl_matrix_memcpy
          > <       0     0.83           1d1990   libgsl.so.23.0.0::gsl_stats_wmean
          > <    1.38     0.77            668e0   libmpi_mt.so::MPI_SGI_request_test
          > <    0.59     0.61            671a0   libmpi_mt.so::MPI_SGI_request_wait
          > <       0     0.54            f3ce0   libgsl.so.23.0.0::gsl_matrix_get_col
          > <    0.01     0.38           12fb00   libc-2.22.so::__memcpy_avx_unaligned
          > <       0     0.32           1d9f60   libgsl.so.23.0.0::gsl_vector_set
          > <       0     0.30           1c5660   libgsl.so.23.0.0::gsl_stats_mean
          > <       0     0.29            f3c30   libgsl.so.23.0.0::gsl_matrix_get_row
```

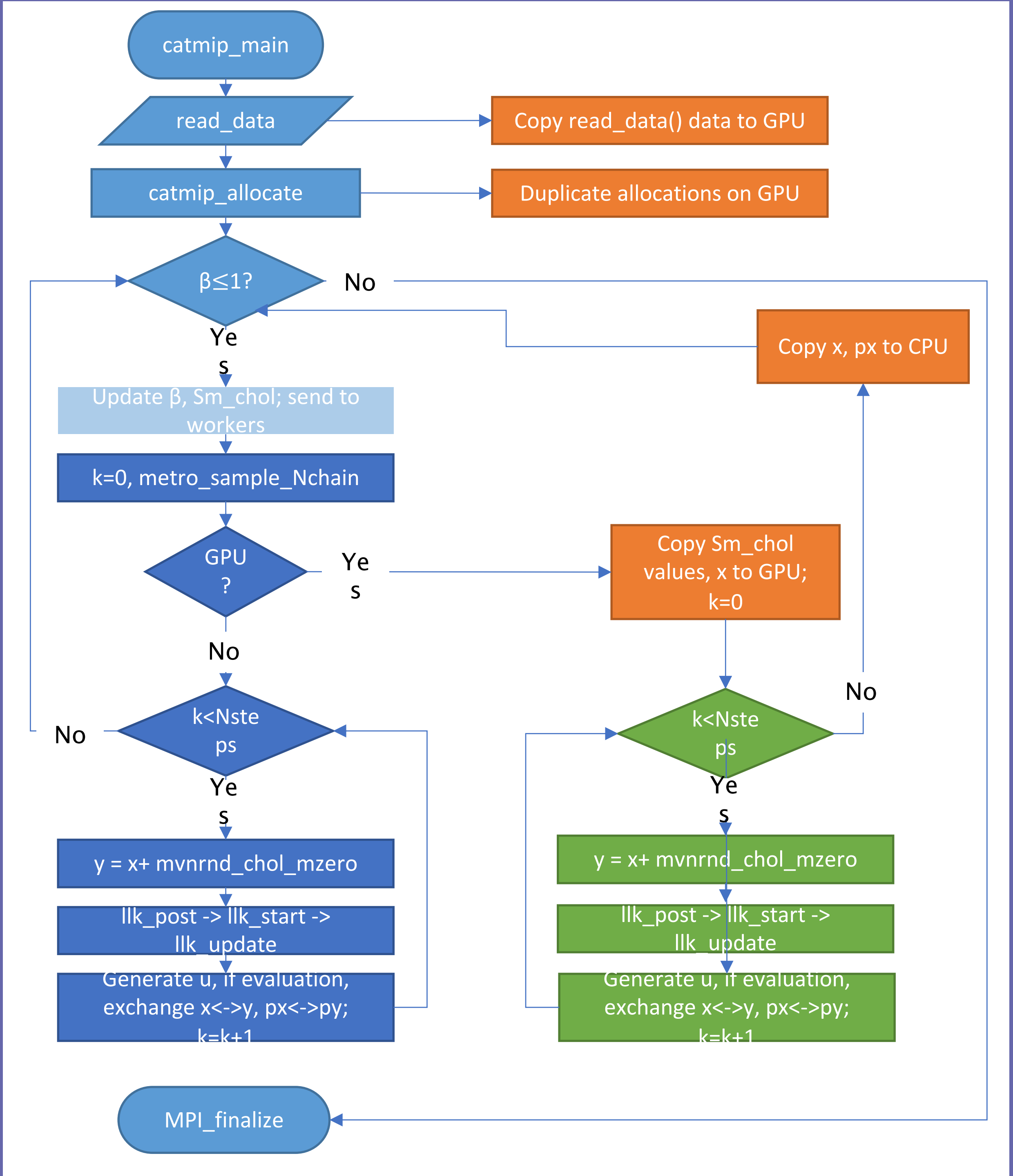
Invoked by GSL for triangular matrix-matrix multiply

- Application is communication intensive
- Computational time spent in calls to the GSL library routines:
  - •Setting and Copying matrices
  - •Operations on upper or lower triangular matrices
    - o GSL calls Intel MKL routines

STRATEGY FOR GPU SUPPORT IN CATMIP

LEGEND

- CPU (master + worker)
- CPU (master only)
- CPU (worker only)
- GPU processing
- GPU-CPU communications

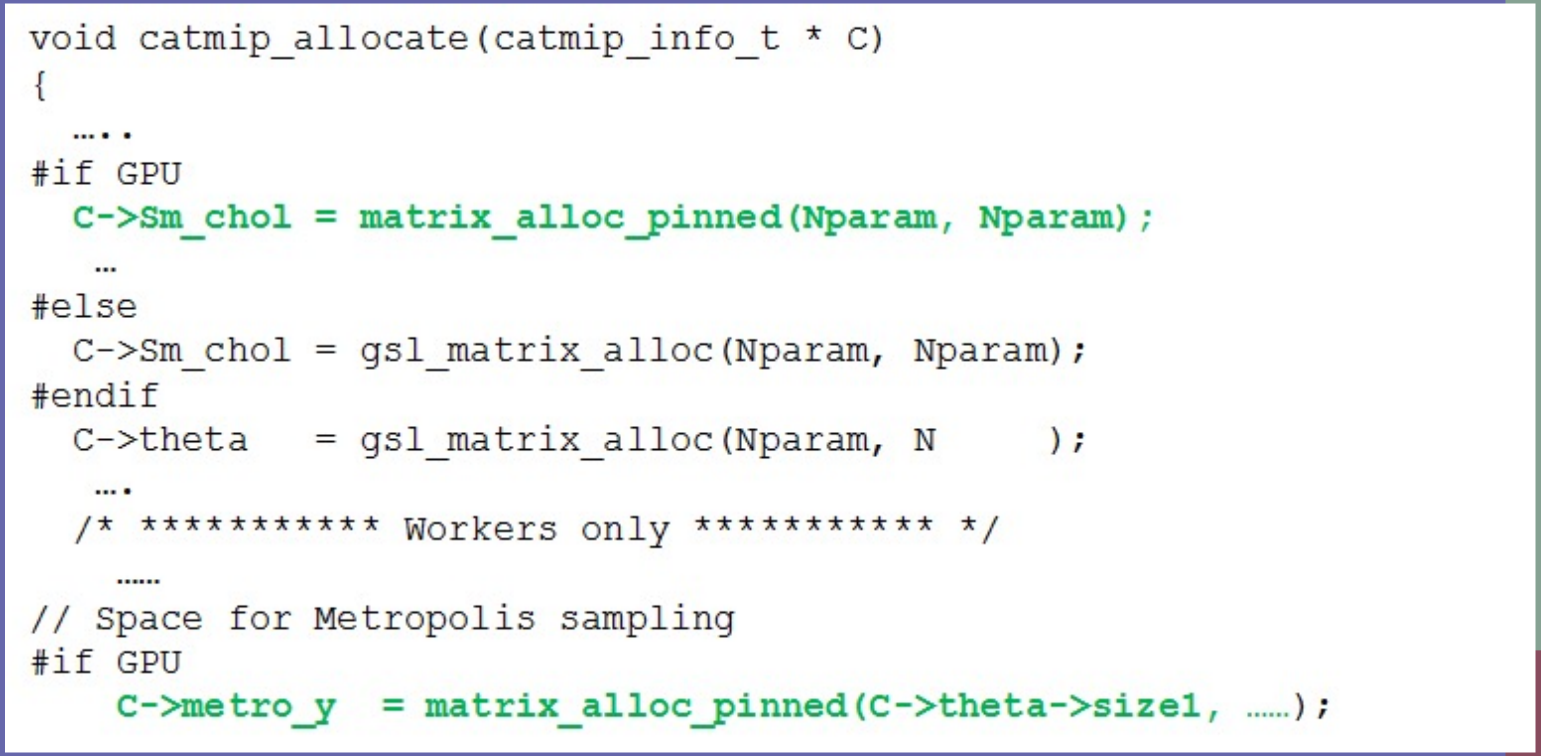

catmip_main
read_data
Copy read_data() data to GPU
catmip_allocate
Duplicate allocations on GPU
β≤1?
No
Copy x, px to CPU
Ye
s
Update β, Sm_chol; send to workers
k=0, metro_sample_Nchain
GPU
?
Ye
s
Copy Sm_chol values, x to GPU; k=0
No
No
k<Nste
ps
No
k<Nste
ps
Ye
s
Ye
s
y = x+ mvnrnd_chol_mzero
y = x+ mvnrnd_chol_mzero
llk_post -> llk_start -> llk_update
llk_post -> llk_start -> llk_update
Generate u, if evaluation, exchange x<->y, px<->py; k=k+1
Generate u, if evaluation, exchange x<->y, px<->py; k=k+1
MPI_finalize

METROPOLIS LOOP WITH GPU SUPPORT

```
#if GPU
    gpu_catmip_copy(C,x,px,sigma); gpu_catmip_memset_accrej();
#endif

for (int k = 0; k < Nsteps; k++) { //for k=1:N0
#if GPU
    matrix_add_gpu((int)(x->size1 * x->size2)); // y = x + N(0,sigma)
#else
    gsl_matrix_add(y,x); // y = x + N(0,sigma)
#endif

    error=llk_post(C,beta,data);//py=LLKfun(y,varargin{:});

//% r = py/px;
#if GPU
    gpu_accept_reject((int)y->size2, (int)y->size1);
#else
for (size_t i=0; i<y->size2; i++) { // Loop over chains
      r = (gsl_matrix_get(py,0,i)-gsl_matrix_get(px,0,i));
      u=gsl_sf_log(my_rand(rng));

      if (u<=r && !error) {
        vv=gsl_matrix_column(py,i); gsl_matrix_set_col(px,i,&vv.vector);
        vv=gsl_matrix_column( y,i); gsl_matrix_set_col( x,i,&vv.vector);
        Naccept++;
      } else {
        Nreject++;
      }
    }
#endif
} // End Nstep loop

#if GPU
  gpu_catmip_copy_back(x->size1, x->size2, x->data, px->data,&Naccept,&Nreject);
#endif
```

CATMIP MEMORY ALLOCATION

- Allocate pinned memory on the host to reduce memory access time
  - Pinned memory on the CPU provides better throughput when copying from device to host than using pageable memory on the CPU
  - Allocating/deallocating pinned memory takes a lot longer than allocation of pageable memory
  - We use pinned CPU memory only for arrays that are allocated just once!
  - Details on allocating a gsl matrix on pinned memory on the next slide
  - https://developer.nvidia.com/blog/how-optimize-data-transfers-cuda-cc/



```
void catmip_allocate(catmip_info_t * C)
{
  …..
#if GPU
  C->Sm_chol = matrix_alloc_pinned(Nparam, Nparam);
   …
#else
  C->Sm_chol = gsl_matrix_alloc(Nparam, Nparam);
#endif
  C->theta    = gsl_matrix_alloc(Nparam, N     );
   ….
  /* *********** Workers only *********** */
    ……
// Space for Metropolis sampling
#if GPU
    C->metro_y  = matrix_alloc_pinned(C->theta->size1, ……);
```

ALLOCATING A GSL MATRIX ON PINNED MEMORY

```
gsl_matrix * matrix_alloc_pinned (const size_t n1, const
size_t n2)
{
  gsl_block * block;
  gsl_matrix * m;
  m =  (gsl_matrix *) malloc (sizeof (gsl_matrix));
  if (m == 0)
    {
      GSL_ERROR_VAL ("failed to allocate space for
matrix struct",
                        GSL_ENOMEM, 0);
    }
  block = block_alloc_pinned (n1 * n2) ;
  if (block == 0)
    {
      GSL_ERROR_VAL ("failed to allocate space for
block",
                        GSL_ENOMEM, 0);
    }
  m->data = block->data;
  m->size1 = n1;
  m->size2 = n2;
  m->tda = n2;
  m->block = block;
  m->owner = 1;
  return m;
}
```

```
gsl_block * block_alloc_pinned (const size_t n)
{
  gsl_block * b;
  b = (gsl_block *) malloc (sizeof (gsl_block));
  if (b == 0)
    {
      GSL_ERROR_VAL ("failed to allocate space for block
struct",
                        GSL_ENOMEM, 0);
    }
  err_cuda = cudaMallocHost((void**)&b->data,n*sizeof(double));
  assert(err_cuda == cudaSuccess);
  if (b->data == 0 && n > 0)
    {
      free (b);/* exception in constructor, avoid memory leak */
      GSL_ERROR_VAL ("failed to allocate space for block data",
                        GSL_ENOMEM, 0);
    }
  b->size = n;
  return b;
}
```

- GSL Matrix is composed of data blocks
- Each data block has to be allocated on pinned memory
- Pinned memory is allocated on the Host via the call to cudaMallocHost

MULTIVARIATE NORMAL RANDOM NUMBER GENERATION

- Matrices Schol and x are stored row-major on the CPU
- Problem: CUBLAS expects col-major
- See following slides for a description how we handled this
- https://docs.nvidia.com/cuda/cublas/index.html

```
int rmvnorm_chol_mzero(const gsl_rng *r,….
…
#if GPU
// assumes that size2, size3 have been set in gpu_catmip_init
// seeding had been done in gpu_catmip_init
gpu_rand (x->size1 * x->size2, seed, x->data);
#else
  for (size_t i=0; i<x->size1; i++) { for (size_t j=0; j<x->size2; j++) {
      gsl_matrix_set(x,i,j, gsl_ran_ugaussian(r)); } } // x <- N(0,1m
#endif
  // CblasLeft: B = alpha*op(A)*B
  // Lower for GSL
#if GPU
  gpu_dtrmm (Schol->size1, x->size1, x->size2, CUBLAS_FILL_MODE_UPPER,….);
#else
  gsl_blas_dtrmm(CblasLeft, CblasLower, CblasNoTrans, …..);
#endif
  // x = chol(S)*x
  return 0;
}
```

CATMIP CUDA LIBRARY EXAMPLES (1)

During initialization:

- Create a handle to the CUBLAS library only once!
- Create and seed the random number generator only once!
- Allocate device memory on the GPU if possible only once
  - Some arrays change sizes and are currently allocated/deallocated

```
extern "C" int gpu_catmip_init(int NR, int NC, int NW)
{….
// create the cublasHandle
   cublasStatus_t err_cublas;
   err_cublas = cublasCreate(&cuHandle);
     `….
/* Create pseudo-random number generator */
    CURAND_CALL(curandCreateGenerator(&cuGen,
                CURAND_RNG_PSEUDO_MT19937));
    /* Set seed */
    CURAND_CALL(curandSetPseudoRandomGeneratorSeed(cuGen,
                1234ULL));
    /* allocate device memory for various arrays

err_cuda = cudaMalloc((void**)&devX,NC*NW*sizeof(double));
```

CATMIP CUDA LIBRARY EXAMPLES (2)

```
extern "C" int gpu_dtrmm(int NR, int NC, int NW, int UPLO, double *B)
{
    cublasStatus_t err_cublas;
    // transfer data: device memory allocated in gpu_catmip_init
    // the memory is populated with numbers by gpu_rand
     const double alpha = 1.0;
      err_cublas = cublasDtrmm(cuHandle, CUBLAS_SIDE_RIGHT,
CUBLAS_FILL_MODE_UPPER, CUBLAS_OP_N, CUBLAS_DIAG_NON_UNIT, NW, NC, &alpha,
devA, NR, devB, NW, devB, NW);
    assert(err_cublas == CUBLAS_STATUS_SUCCESS);
    return EXIT_SUCCESS;
}
```

- Data for devA and devB is row major in memory, CUBLAS expects col-major
- We trick CUBLAS into thinking that the matrices are col-major by choosing:
- CUBLAS_SIDE_RIGHT: we have to put A on the right, since the dimensions in A and B that have to line up have been transposed ((AB)^T = (B^T)(A^T))
- CUBLAS_FILL_MODE_UPPER: bottom row is all zeros except last entry,
- CUBLAS_OP_N: Due to the col-major storage we do not transpose but switch dimensions of the matrix devB and adjust leading dimension accordingly
- Note that the leading dimension of devB remains NW for row and col storage

CATMIP CUDA LIBRARY EXAMPLES (3)

```
extern "C" int gpu_rand (int n, double seed, double *hostData)
{
    /* Create pseudo-random number generator */
    /* handled by gpu_catmip_init */
    /* Allocate n double on device: this is devB allocated in gpu_catmip_init */
    /* Generate n doubles on device directly into output matrix */
      CURAND_CALL(curandGenerateNormalDouble(cuGen, devB, n, 0.0, 1.0));
    /* Copy device memory to host: Not necessary, data will be used gpu_dtrmm */
    /* Cleanup handled in gpu_catmip_cleanup*/
    return EXIT_SUCCESS;
}
```

CATMIP CUDA LIBRARY EXAMPLES (4)

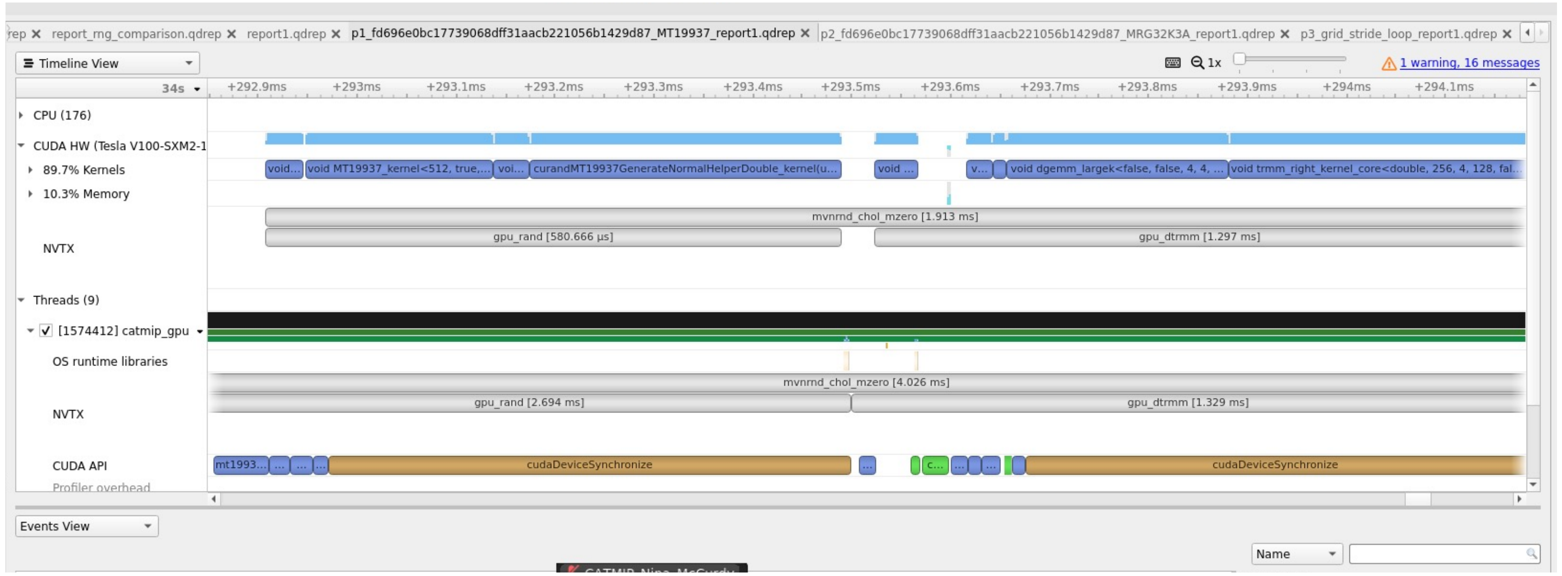
```
__global__ void gpu_accept_reject_kernel (int size, double * devPx, double *
devPy, double * devR, double * devU, int Nparam, double * devC, double *
devB, int * devNaccept, int * devNreject)
{
 // precalculate random numbers and conditional
    int index = threadIdx.x + blockIdx.x * blockDim.x;
    int j = index;
    if (index < size) {
        devR[index] = devPy[index] - devPx[index];
        devU[index] = log((double) devU[index]);
        if (devU[index]<=devR[index]) { // accept
          for (int i=0; i<3; i++) {
            devPx[j+i*size] = devPy[j+i*size]; }
          for (int i=0; i<Nparam; i++) {
            devC[j+i*size]=devB[j+i*size]; }
          atomicAdd(devNaccept,1);
        } else { // reject
          atomicAdd(devNreject,1);
        }
     }
    return;
}
```

- Atomic updates were necessary for correct execution

Optimizing Random Number Generation

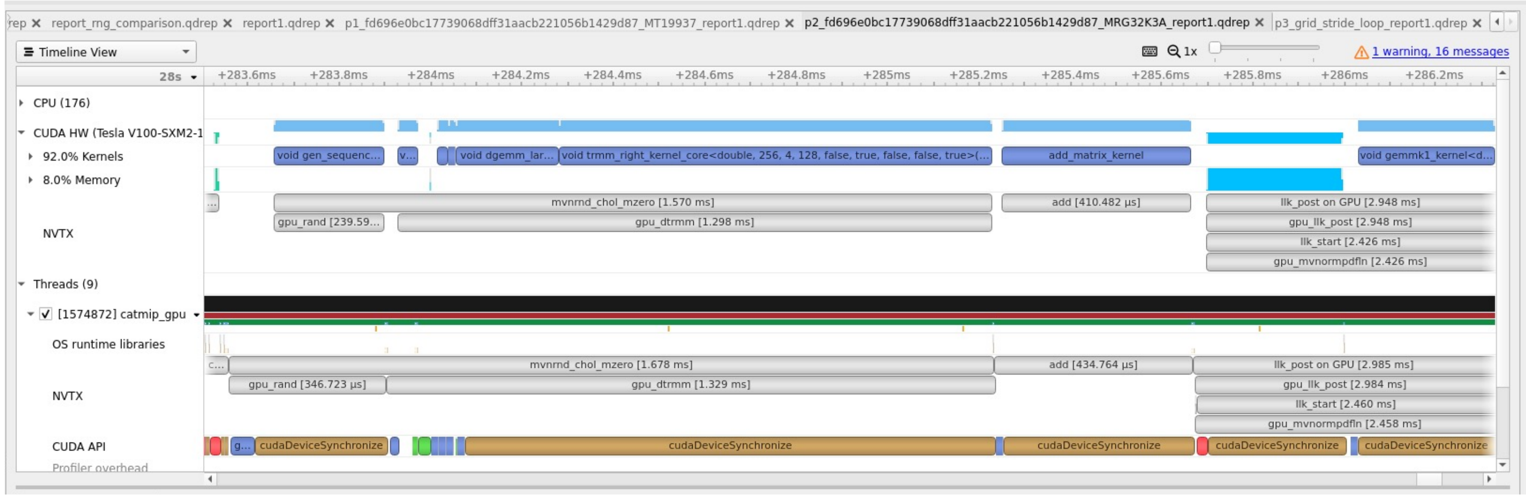
Original gpu_rand (MT19937)

580.666 µS



rep
report_rng_comparison.qdrep
report1.qdrep
p1_fd696e0bc17739068dff31aacb221056b1429d87_MT19937_report1.qdrep
p2_fd696e0bc17739068dff31aacb221056b1429d87_MRG32K3A_report1.qdrep
p3_grid_stride_loop_report1.qdrep
Timeline View
1x
1 warning, 16 messages
34s
+292.9ms
+293ms
+293.1ms
+293.2ms
+293.3ms
+293.4ms
+293.5ms
+293.6ms
+293.7ms
+293.8ms
+293.9ms
+294ms
+294.1ms
CPU (176)
CUDA HW (Tesla V100-SXM2-1
89.7% Kernels
10.3% Memory
NVTX
void...
void MT19937_kernel<512, true,...
voi...
curandMT19937GenerateNormalHelperDouble_kernel(u...
void ...
v...
void dgemm_largek<false, false, 4, 4, ...
void trmm_right_kernel_core<double, 256, 4, 128, fal...
mvnrnd_chol_mzero [1.913 ms]
gpu_rand [580.666 µs]
gpu_dtrmm [1.297 ms]
Threads (9)
[1574412] catmip_gpu
OS runtime libraries
mvnrnd_chol_mzero [4.026 ms]
gpu_rand [2.694 ms]
gpu_dtrmm [1.329 ms]
NVTX
CUDA API
mt1993...
cudaDeviceSynchronize
c...
cudaDeviceSynchronize
Profiler overhead
Events View
Name

New gpu_rand (MRG32K3A) 239.594 µs



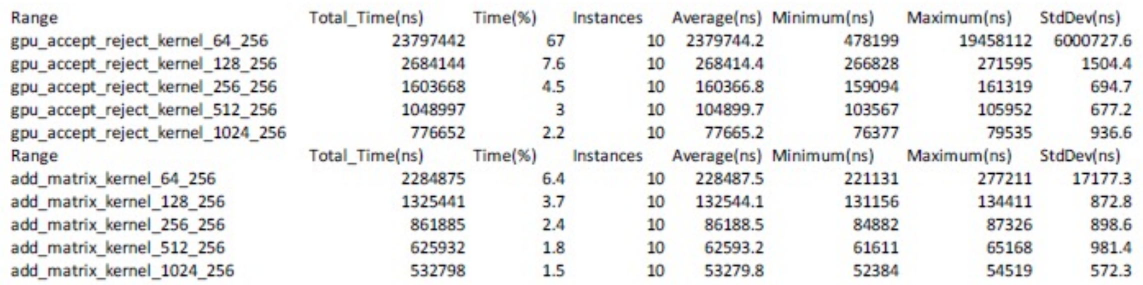
Using Striding in CUDA Kernels

```
__global__ void add_matrix_kernel (double *b, double *a, int n) {
  for (int index = blockIdx.x * blockDim.x + threadIdx.x; index < n; index += blockDim.x * gridDim.x) {
      b[index] = b[index] + a[index];
  }
```

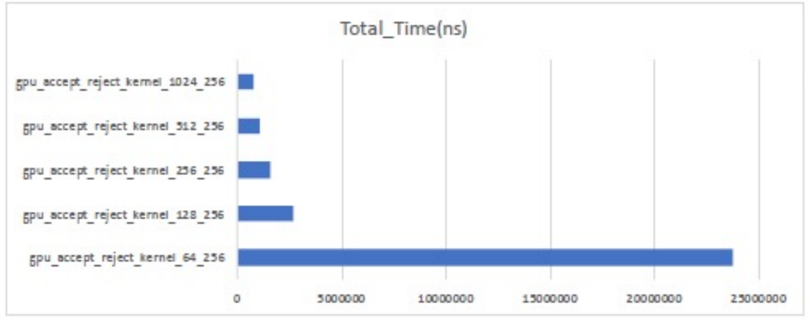
```
extern "C" void matrix_add_gpu (int nsize)
{
    add_matrix_kernel<<<(nsize+TPB-1)/TPB,TPB>>> (devB, devC, (int)nsize);
 }
  return;
}
```

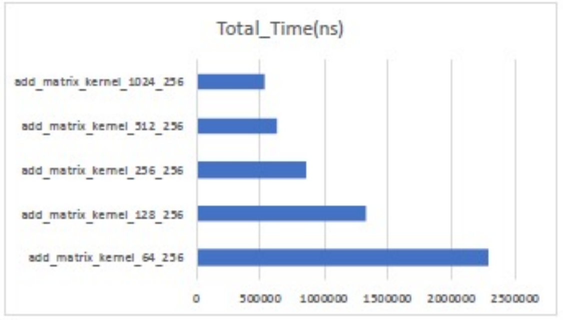
Threads per block

Number of blocks



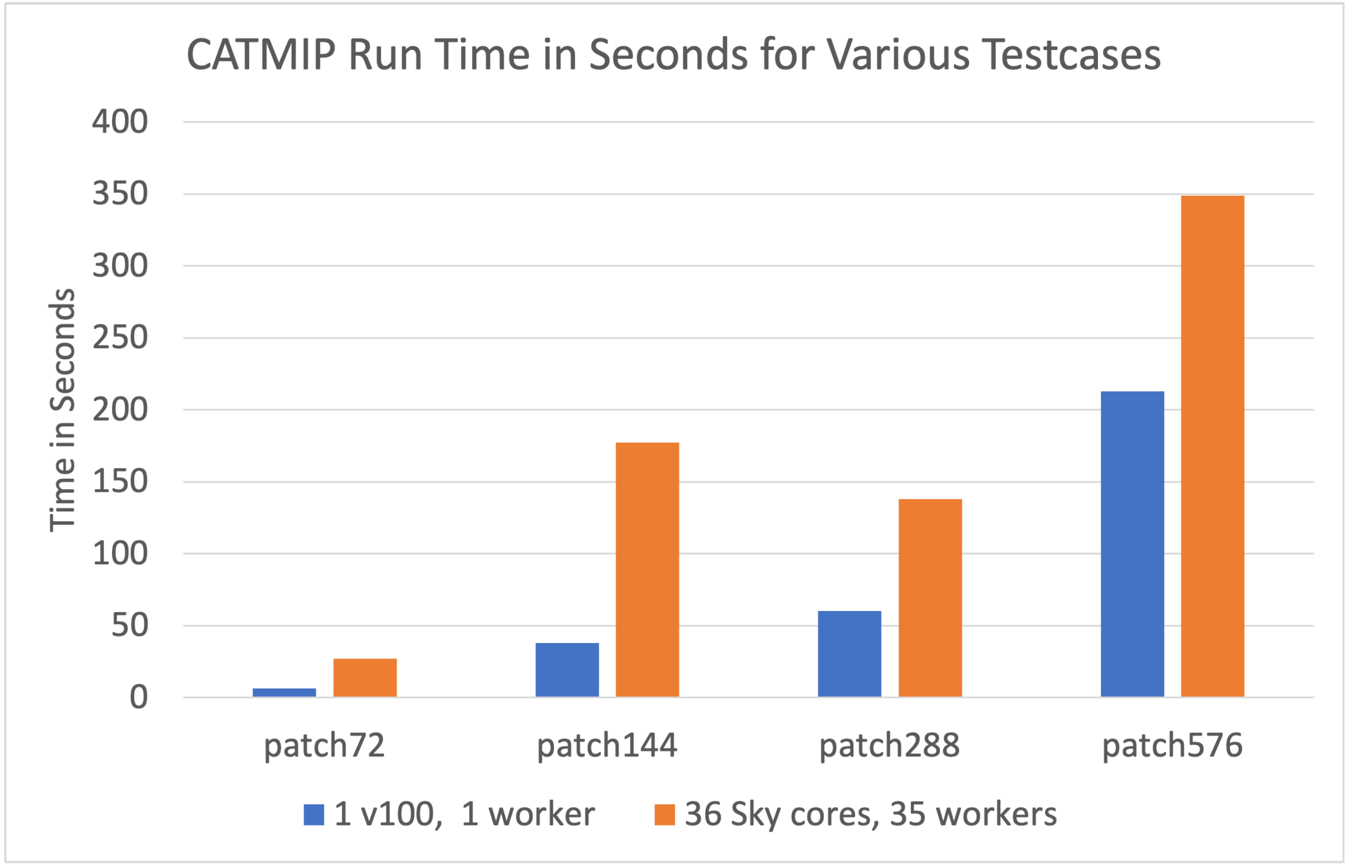
Range	Total_Time(ns)	Time(%)	Instances	Average(ns)	Minimum(ns)	Maximum(ns)	StdDev(ns)
gpu_accept_reject_kernel_64_256	23797442	67	10	2379744.2	478199	19458112	6000727.6
gpu_accept_reject_kernel_128_256	2684144	7.6	10	268414.4	266828	271595	1504.4
gpu_accept_reject_kernel_256_256	1603668	4.5	10	160366.8	159094	161319	694.7
gpu_accept_reject_kernel_512_256	1048997	3	10	104899.7	103567	105952	677.2
gpu_accept_reject_kernel_1024_256	776652	2.2	10	77665.2	76377	79535	936.6
Range	Total_Time(ns)	Time(%)	Instances	Average(ns)	Minimum(ns)	Maximum(ns)	StdDev(ns)
add_matrix_kernel_64_256	2284875	6.4	10	228487.5	221131	277211	17177.3
add_matrix_kernel_128_256	1325441	3.7	10	132544.1	131156	134411	872.8
add_matrix_kernel_256_256	861885	2.4	10	86188.5	84882	87326	898.6
add_matrix_kernel_512_256	625932	1.8	10	62593.2	61611	65168	981.4
add_matrix_kernel_1024_256	532798	1.5	10	53279.8	52384	54519	572.3





- In CUDA, the number of blocks and the number of threads per block are parameters that affect performance without having to change any code.
- Adding a striding mechanism makes your code future proof:
  - You can experiment with the launch configuration in the future without having to rewrite the kernel again
  - Best practice is to start with the striding in any new kernels
- An example on how the number of blocks can impact the performance in chart on the left:
  - We left the TPB constant at 256
  - The number of blocks varied from 64 to 1024
  - A higher number of blocks was beneficial

CATMIP Timings on CPU and GPU



- We collected timings for a variety of test cases
- GPU timings were collected using 1 V100 GPU and one worker process
- CPU timings were collected using 1 Skylake node and 35 worker processes
- Cases patch72 and patch144 employed 1 Markov Chain length of 1000; patch 288 and patch 576 employed a length of 100
- The GPU was particularly beneficial for long Markov Chains

Results were obtained at NASA Ames Research Center
https://www.nas.nasa.gov/hecc/support/kb/using-gpu-nodes_298.html

CONCLUSIONS

- CATMIP is an efficient sampler for generating samples of a target probability density function
- We have used BLAS and GPU acceleration to optimize both the CATMIP library itself and a user library for earthquake slip modeling
- With GPU acceleration, we can now run in a variety of environments from large CPU clusters to single GPU accelerated servers and everything in between