Kernel Tuner Tutorial – Intermediate Topics

netherlands eScience center

Intermediate topics session outline

- GPU Optimizations
- Output verification
- Search space restrictions
- Caching tuning results
- Third hands-on session
- Break

GPU optimizations

GPU code optimizations

- Modify the kernel code in an attempt to improve performance or tunability
- Effects on performance can be different on different GPUs or different input data
- You can tune
  - enabling or disabling an optimization
  - the parameters introduced by certain optimizations
- You often need to combine multiple different optimizations with specific tunable parameter values to arrive at optimal performance

Overview of GPU Optimizations

- Coalescing memory accesses
- Host/device communication
- Kernel fusion
- Loop blocking
- Loop unrolling
- Prefetching
- Recomputing values
- Reducing atomics
- Reducing branch divergence
- Reducing redundant work
- Reducing register usage
- Reformatting input data
- Using a specific memory space
- Using warp shuffle instructions
- Varying work per thread
- Vectorization

Overview of GPU Optimizations

- Coalescing memory accesses
- Host/device communication
- Kernel fusion
- Loop blocking
- **Loop unrolling**
- Prefetching
- Recomputing values
- Reducing atomics
- Reducing branch divergence
- Reducing redundant work
- **Reducing register usage**
- Reformatting input data
- Using a specific memory space
- Using warp shuffle instructions
- **Varying work per thread**
- **Vectorization**

(Partial) Loop Unrolling

- Why?
  - Increases instruction-level-parallelism
  - Reduces loop overhead instructions
- How?
  - In the early days, only manually or with a code generator
  - Compiler does this now: `#pragma unroll <value>`
  - In CUDA, value has to be integer constant expression
    - 0 is not allowed, 1 means unrolling is disabled
  - Remember, Kernel Tuner inserts parameters with `#define`
  - Parameters that start with `loop_unroll_factor_` are inserted as integer constant expressions instead, on 0 KT removes line with pragma

Partial loop unrolling

```
...

#pragma unroll loop_unroll_factor_nlay
for (int ilay=0; ilay<nlay; ++ilay) {
    ...
}
```

The compiler can unroll this loop if `nlay` is known at compile-time. The `loop_unroll_factor_nlay` parameter should be a divisor of `nlay`.

Reducing register usage

- Why?
  - Registers are an important and limited SM resource and are likely to limit occupancy
  - Allows to increase the tunable range of thread block dimensions
- How?
  - Compiling constant values into your code rather than keeping them in registers (e.g. using templates or tunable parameters)
  - Limiting or disabling loop unrolling are very effective ways of reducing register usage
  - In kernels that do many different things, splitting the kernel may help to cut down register usage
  - Enabling register spilling with compiler flag `-maxrregcount=N`

Reducing register usage

```
template<typename TF>__global__
void some_kernel(const int ncol,
                 const int nlay,
                 const int ngpt,
                 const int top_at_1, TF* flux_dn)
{
    const int icol = blockIdx.x*blockDim.x + threadIdx.x;
    const int igpt = blockIdx.y*blockDim.y + threadIdx.y;
    if ( (icol < ncol) && (igpt < ngpt) )
    {
        if (top_at_1)
        {
            ...
        }
        else
        {
            ...
        }
    }
}
```

```
template<typename TF, int top_at_1>__global__
void some_kernel(const int ncol,
                 const int nlay,
                 const int ngpt,
                 TF* flux_dn)
{
    const int icol = blockIdx.x*blockDim.x + threadIdx.x;
    const int igpt = blockIdx.y*blockDim.y + threadIdx.y;
    if ( (icol < ncol) && (igpt < ngpt) )
    {
        if (top_at_1)
        {
            ...
        }
        else
        {
            ...
        }
    }
}
```

Varying work per thread

- Why?
  - Increasing work per thread often increases data reuse and locality
  - Reduces redundant instructions previously executed by other threads
  - Increases instruction-level parallelism and possibly increases register usage
- How?
  - Reduce number of threads blocks in total, but increase the work per thread block
  - Bring down number of threads within the block, but keep the amount of work equal

Varying work per thread

```
...
#pragma unroll
for (kb = 0; kb < block_size_x; kb++) {

    sum[i][j] += sA[ty][kb] * sB[kb][tx];

}
```

```
...
#pragma unroll
for (kb = 0; kb < block_size_x; kb++) {

    #pragma unroll
    for (int j = 0; j < tile_size_x; j++) {
        sum[i][j] += sA[ty][kb] * sB[kb][tx + j * block_size_x];
    }

}
```

Vectorization

- Why?
  - Reduces the instructions needed to fetch data from global memory
  - Improves memory throughput
  - Often also increases work per thread and instruction-level parallelism
  - May increase register usage
- How?
  - Using wider data types (e.g. `float2` or `float4` instead of `float`)
  - Vector length can be tuned

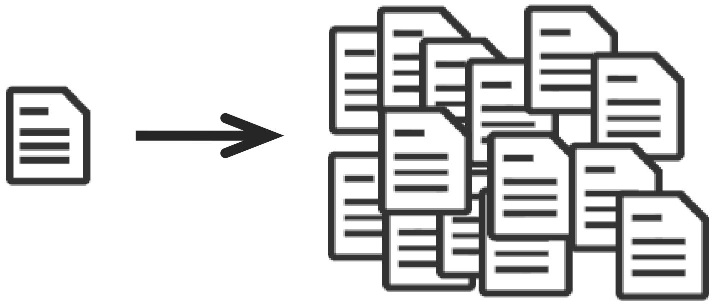
Vectorization

```
#if (vector==1)
#define floatvector float
#elif (vector == 2)
#define floatvector float2
#elif (vector == 4)
#define floatvector float4
#endif

__global__ void sum_floats(float *sum_global, floatvector *array, int n) {

    int x = blockIdx.x * block_size_x + threadIdx.x;
    if (x < n/vector) {
        floatvector v = array[x];

        ...
    }
...
```

Output verification

Programming tunable applications

- When working with tunable code you are essentially maintaining many different versions of the same program in a single source
- It may happen that certain combinations of tunable parameters lead to versions that produce incorrect results
- Kernel Tuner can verify the output of kernels while tuning!



Output verification

- When you pass a reference `answer` to `tune_kernel`:
  - Kernel Tuner will run the kernel once before benchmarking and compare the kernel output against the reference `answer`
  - The `answer` is a list that matches the kernel arguments in number, shape, and type, but contains None for input arguments
  - By default, Kernel Tuner will use `np.allclose()` with an absolute tolerance of `1e-6` to compare the state of all kernel arguments in GPU memory that have non-None values in the `answer` array
- And of course, you can modify this behavior, but first a simple example

Simple answer example

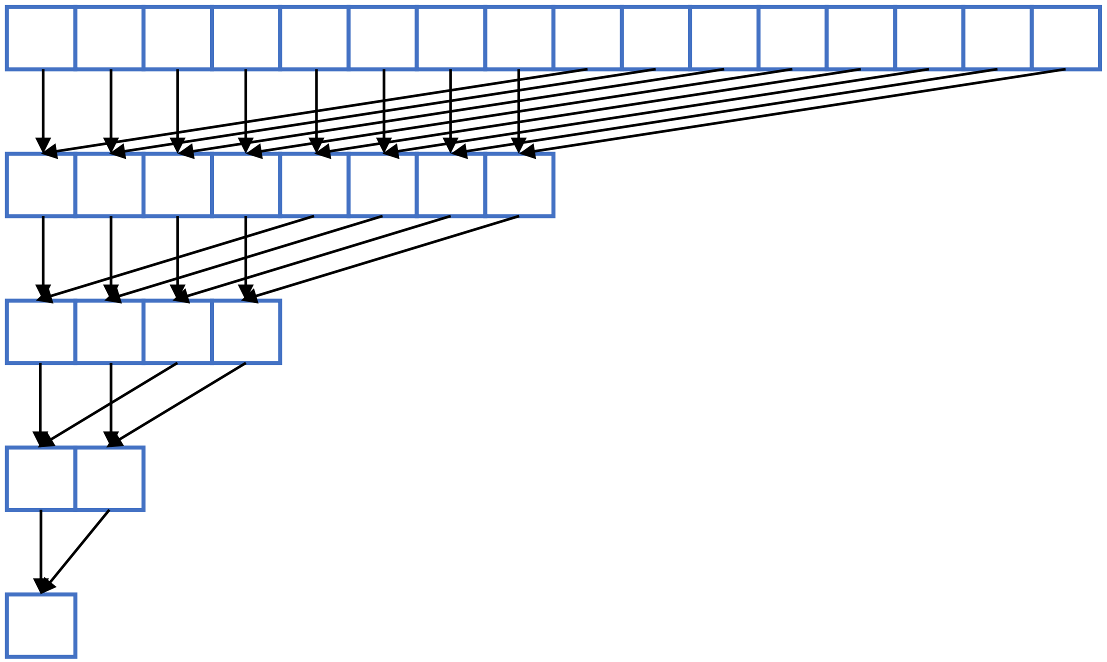
```
args = [c, a, b, n]
answer = [a+b, None, None, None]

tune_kernel("vector_add", kernel_string, size, args, tune_params,
            answer=answer, atol=1e-3)
```

Custom verify functions

- For some kernels the default verification functionality is not enough
- For example when output is different for different tunable parameters
- You can pass a function to the `verify` optional argument of `tune_kernel()`
- The verify function should take 3 arguments: a reference, the result, and a tolerance

Custom verify example - reduction

- Say we have reduction kernel in which all thread blocks as a group iterate over the input
- Then each thread block computes a thread-block-wide partial sum
- A second kernel is used to sum all partial sums to a single summed value



Custom verification function - wrong

```
tune_params["block_size_x"] = [32, 64, 128, 256, 512, 1024]
tune_params["num_blocks"] = [32, 64, 128, 256, 512, 1024, 2048, 4096, 8192, 16384, 32768]
problem_size = "num_blocks"

args = [sum_x, x, n]
reference = [numpy.sum(x), None, None]

tune_kernel("sum_floats", kernel_string, problem_size, args, tune_params, grid_div_x=[],
            verbose=True, answer=reference)
```

Custom verification function

```
tune_params["block_size_x"] = [32, 64, 128, 256, 512, 1024]
tune_params["num_blocks"] = [32, 64, 128, 256, 512, 1024, 2048, 4096, 8192, 16384, 32768]
problem_size = "num_blocks"


args = [sum_x, x, n]
reference = [numpy.sum(x), None, None]

def verify_partial_reduce(cpu_result, gpu_result, atol=None):
    return numpy.isclose(cpu_result[0], numpy.sum(gpu_result[0]), atol=atol)


tune_kernel("sum_floats", kernel_string, problem_size, args, tune_params, grid_div_x=[],

            verbose=True, answer=reference, verify=verify_partial_reduce)
```

Search space restrictions

Restricting the search

- By default, the search space is the Cartesian product of all possible combinations of tunable parameter values
- Example:

```
tune_params["block_size_x"] = [32, 64, 128, 256, 512]
tune_params["vector"] = [1, 2, 4]
tune_params["use_shared_mem"] = [0, 1]
```

- However, for some tunable kernels:
  - there are tunable parameters that depend on each other
  - only certain combinations of tunable parameter values are valid

Dependent parameters example

- In this example:
  - We have a parameter that controls a loop count, `tile_size_x`
  - And we want to also tune the partial loop unrolling factor of that loop, using a parameter named `loop_unroll_factor_x`
- Kernel Tuner considers the Cartesian product of all possible values of both parameters as the search space
- But only configurations in which `loop_unroll_factor_x` is a divisor of `tile_size_x` are valid

Partial loop unrolling example

```
tune_params["tile_size_x"] = [1, 2, 3, 4, 5, 6, 7, 8]
tune_params["loop_unroll_factor_x"] = [1, 2, 3, 4, 5, 6, 7, 8]

restrictions = lambda p: p["loop_unroll_factor_x"] <= p["tile_size_x"] and
                         p["tile_size_x"] % p["loop_unroll_factor_x"] == 0
```

Caching tuning results

Caching

- Tuning large search spaces can take very long
- You might need to stop and continue later on
- Caching is enabled by passing a filename to the **cache** option
- Kernel Tuner will append new results to the cache directly after benchmarking a kernel configuration
- Kernel Tuner detects existing (possibly incomplete) cache files and automatically resumes tuning where it had left off

Simulation runner

- In the next part of this tutorial, we will look into using optimization strategies
- Cache files can also be used to quickly benchmark different optimization strategies or tune hyperparameters
- To use the simulation runner set `simulation_mode=True` with an existing `cache` file that contains information on ***all*** configurations in the search space

Third hands-on session

Intermediate hands-on

- The third hands-on notebook is:
  - https://github.com/benvanwerkhoven/kernel_tuner_tutorial/blob/master/hands-on/cuda/02_Kernel_Tuner_Intermediate.ipynb
- The goal of this hands-on is to experiment with search space restrictions, caching, and output verification
  - Copy the notebook to your Google Colab and work there
- Please follow the instructions in the Notebook
- Feel free to ask questions to instructors and mentors

Optional hands-on

- Done with the third hands-on already?
- Keep playing with this notebook
  - https://github.com/benvanwerkhoven/kernel_tuner_tutorial/blob/master/hands-on/cuda/Kernel_Tuner_Tutorial.ipynb
- Keep experimenting with your own code
- Feel free to ask questions to instructors and mentors