Autotuning

Introduction to autotuning, overview of our research

Jiří Filipovič et al.
Institute of Computer Science
Masaryk University

2019

Program development workflow

Implementation questions

- which algorithm to use?
- how to implement the algorithm efficiently?
- how to set-up a compiler?

Program development workflow

Compiler's questions

- how to map variables to registers?
- which unrolling factor to use for a loop?
- which functions should be inlined?
- and many others...

Program development workflow

Execution

- how many nodes and threads assign to the program?
- should accelerators be used?
- how to mix MPI and OpenMP threads?

Program development workflow

Execution

- how many nodes and threads assign to the program?
- should accelerators be used?
- how to mix MPI and OpenMP threads?

A compiler works with **heuristics**, people usually too.

Tuning of the program

We can empirically tune those possibilities

- use different algorithm
- change code optimizations
- use different compiler flags
- execute in a different number of threads
- etc.

Tuning of the program

A tuning allows us to outperform heuristics – we just test what works better.

- however, we have to invest more time into development
- there are vertical dependencies, so we cannot perform tuning steps in isolation
- the optimum usually **depends on hardware and input**

Autotuning

The tuning can be automated

- then we talk about **autotuning**

Autotuning

- in design time, we define the space of *tuning parameters*, which can be changed
- each tuning parameter defines some property of the tuned application
- a search method is used to traverse the space of tuning parameters efficiently
- performed according to some objective, usually performance

Taxonomy of Autotuning

Tuning scope

- what properties of the application are changed by autotuner
- e.g. compiler flags, number of threads, source code optimizations parameters

Tuning time

- off-line autotuning (performed once, e.g. after SW installation)
- dynamic autotuning (performed in runtime)

Developer involvement

- transparent, or requiring only minor developer assist (e.g. compiler flags tuning)
- low-level, requiring an expert programmer to identify tunning opportunities (e.g. optimizations parameters tuning)

Our focus

We target autotuning of code optimization parameters

- the source code is changed during a tuning process
- the user defines how tuning parameters influence the code
- very powerful (source code may control nearly everything)
- implementation is difficult
  - requires recompilation
  - runtime checks of correctness/precision
  - non-trivial expression of tuning parameters
  - we have no implicit assumptions about tuning space
- heterogeneous computing (we are tuning OpenCL or CUDA code)
- offline and dynamic autotuning

Motivation Example

Let's solve a simple problem – vectors addition

- we will use CUDA
- we want to optimize the code

Motivation Example

```
__global__ void add(float* const a, float* b) {
    int i = blockIdx.x*blockDim.x + threadIdx.x;
    b[i] += a[i];
}
```

It should not be difficult to write different variants of the code...

Optimization

```
__global__ void add(float4* const a, float4* b) {
    int i = blockIdx.x*blockDim.x + threadIdx.x;
    b[i] += a[i];
}
```

Kernel has to be executed with n/4 threads.

Optimization

```
__global__ void add(float2* const a, float2* b) {
    int i = blockIdx.x*blockDim.x + threadIdx.x;
    b[i] += a[i];
}
```

Kernel has to be executed with n/2 threads.

Optimization

```
__global__ void add(float* const a, float* b, const int n) {
    int i = blockIdx.x*blockDim.x + threadIdx.x;
    for (; i < n; i += blockDim.x*gridDim.x)
        b[i] += a[i];
}
```

Kernel has to be executed with n/m threads, where *m* can be anything.

What to Optimize?

Mixture of:

- thread-block size
- vector variables
- serial work

i.e. 3D space – and this is trivial example...

Autotuning

Autotuning tools may explore code parameters automatically

```
__global__ void
add(VECTYPE* const a, VECTYPE* b, const int n) {
    int i = blockIdx.x*blockDim.x + threadIdx.x;
#if SERIAL_WORK > 1
    for (; i < n; i += blockDim.x*gridDim.x)
#endif
        b[i] += a[i];
}
```

The code executing kernel `add` has to configure parallelism according to values of `VECTYPE` and `SERIAL_WORK` tuning parameters.

Kernel Tuning Toolkit

We have developed a Kernel Tuning Toolkit (KTT)

- a framework allowing to tune code parameters for OpenCL and CUDA
- allows both offline and dynamic tuning
- enables cross-kernel optimizations
- mature implementation, documented, with examples
- `https://github.com/Fillo7/KTT`

Kernel Tuning Toolkit

Typical workflow similar to CUDA/OpenCL

- initialize the tuner for a specified device
- create input/output of the kernel
- create kernel
- create a tuning space for the kernel
- assign input/output to the kernel
- execute or tune the kernel

KTT creates a layer between an application and OpenCL/CUDA.

KTT Sample Code

```
// Initialize tuner and kernel
ktt::Tuner tuner(platformIndex, deviceIndex);
const ktt::DimensionVector ndRangeDimensions(inputSize);
const ktt::DimensionVector workGroupDimensions(128);
ktt::KernelId foo = tuner.addKernelFromFile(kernelFile, "foo",
  ndRangeDimensions, workGroupDimensions);

// Creation and assign of kernel arguments
ktt::ArgumentId a = tuner.addArgumentVector(srcA,
  ktt::ArgumentAccessType::ReadOnly);
ktt::ArgumentId b = tuner.addArgumentVector(srcB,
  ktt::ArgumentAccessType::WriteOnly);
tuner.setKernelArguments(foo,
  std::vector<ktt::ArgumentId>{a, b});

// Addition of tuning variables
tuner.addParameter(foo, "UNROLL", {1, 2, 4, 8});

tuner.tuneKernel(foo);
tuner.printResult(foo, "foo.csv", ktt::PrintFormat::CSV);
```

Kernel Tuning Toolkit

In practise, we usually need more functionality

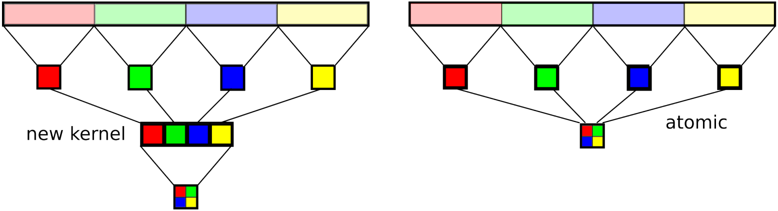
- tuning parameters can affect parallelism configuration (e.g. block and grid size in CUDA)
  - by pre-defined functions (e.g. multiply specified block/grid dimmension)
  - by lambda function provided by programmer
- some combinations of tuning parameters can be discarded *a priori*
  - lambda functions constraining tuning space
- KTT can check, if tuned kernel runs successfully
  - automatic check of successful execution
  - user can provide reference kernel, or reference class, and comparing function, KTT compares results automatically

Advanced features of KTT

Cross-kernel optimizations

- the user can add specific code for kernels execution into `launchComputation` method
- the code may query tuning parameters
- the code may call multiple kernels
- allows tuning code parameters with wider influence, as tuned kernels do not need to be functionally equivalent

Reduction



Advanced features of KTT

Dynamic autotuning

- dynamic tuning performs autotuning during application runtime
- KTT can execute the best kernel known so far to perform kernel's task
- or try different combination of tuning parameters before the execution
- tuning is transparent for the application
- tuning can be queried in any time

Dynamic Tuning Sample

```
// Main application loop
while(application_run) {
  ...
  if (tuningRequired)
    tuner.tuneKernelByStep(foo, {b});
  else {
    ktt::ComputationResult best =
      tuner->getBestComputationResult(foo);
    tuner.runKernel(compositionId,
      best.getConfiguration(), {b});
  }
  ...
}
```

Dynamic tuning

Dynamic autotuning is challenging

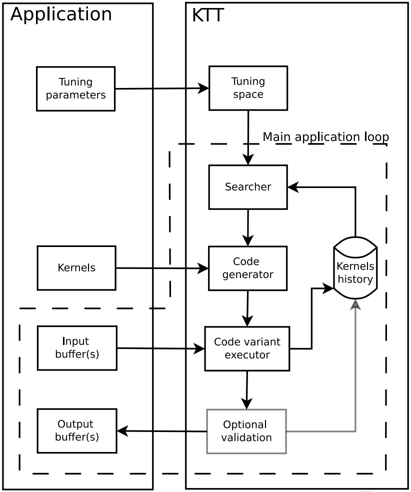
- when the kernel is executed, there must be no significant performance drop
- automatic memory management has to move only necessary data
- KTT has to support asynchronous execution of
  - memory copy, host and device code execution
  - simultaneous execution of multiple kernels

Parallelism in KTT

- intra-manipulator: parallelism inside `launchComputation` method
- global parallelism: asynchronous execution of multiple `launchComputation` instances

During autotuning, global parallelism is disabled.

KTT Architecture



Benchmark set

Benchmark	dimensions	configurations
BiCG	11	5,122
Convolution	10	5,248
Coulomb 3D	8	1,260
GEMM	15	241,600
GEMM batched	11	424
Hotspot	6	480
Transpose	9	10,752
N-body	8	9,408
Reduction	5	175
Fourier	6	360

Table: A list of the benchmarks and the size and dimensionality (i.e., the number of tuning parameters) of their tuning spaces.

Testbed setup

Device	Architecture	SP perf.	BW
2× Xeon E5-2650	Sandy Bridge	512	102
Xeon Phi 5110P	Knights Corner	2,022	320
Tesla K20	Kepler	3,524	208
GeForce GTX 750	Maxwell	1,044	80
GeForce GTX 1070	Pascal	5,783	256
Radeon RX Vega 56	GCN 5	8,286	410
GeForce RTX 2080Ti	Turing	11,750	616

Table: Devices used in our benchmarks. Arithmetic performance (SP perf.) is measured in single-precision GFlops, memory bandwidth (BW) is measured in GB/s.

Performance

Benchmark	2080Ti	1070	750	K20	Vega56	E5-2650	5110P
BiCG	88.3%	84.7%	81.7%	50.4%	75.6%	46.0%	6.45%
Coulomb 3D	91.8%	91.4%	84.3%	43.2%	65.3%	74.2%	22.2%
GEMM	79.8%	80.6%	91.1%	51.3%	96.3%	37.5%	19.7%
GEMM batched	86.8%	81.4%	90.0%	49.6%	86.0%	27.7%	20.9%
Transpose	87.1%	80.2%	86.3%	64.2%	86.1%	62.5%	10.0%
N-body	89.7%	86.6%	87.7%	40.6%	82.2%	77.7%	29.9%
Reduction	68.7%	87.5%	89.4%	64.1%	71.6%	33.9%	10.1%
Hotspot	1.35×	1.94×	2.06×	1.4×	2.88×	1.2×	12.8×

Table: Performance of benchmarks autotuned for various hardware devices. The performance relative to the theoretical peak of devices.

Performance portability

	GPU→GPU		
Benchmark	avg±stdev	worst	failed
BiCG	89.0%±12.3%	57%	1
Convolution	79.4%±14.9%	55%	3
Coulomb 3D	95.8%±6.5%	67%	0
GEMM	83.6%±16.4%	31%	0
GEMM batched	85.4%±17%	37%	0
Hotspot	80.3%±17.5%	46%	3
Transpose	85.0%±21.9%	8%	3
N-body	78.8%±24.2%	2%	3
Reduction	88.4%±24%	12%	3
Fourier	74.5%±30%	31%	0

Table: Relative performance of benchmarks ported across GPU architectures without re-tuning.

Dynamic autotuning of Batched GEMM

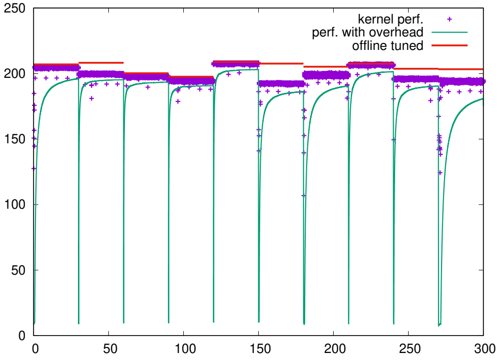


Figure: Batched GEMM on GeForce GTX 1070.

Dynamic autotuning of Batched GEMM

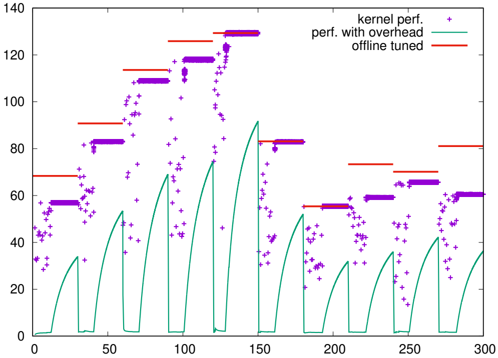


Figure: Batched GEMM on Tesla K20.

3D Fourier Reconstruction

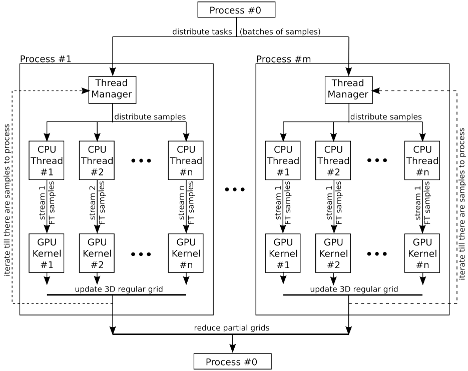


Figure: Performance of dynamic tuned 3D Fourier reconstruction.

3D Fourier Reconstruction

	2080Ti	1070	750	680
2080Ti	100%	99%	31%	49%
1070	99%	100%	31%	50%
750	43%	67%	100%	94%
680	60%	72%	71%	100%

Table: Performance portability of 3D Fourier reconstruction with 128×128 samples.

3D Fourier Reconstruction

	128x128	91x91	64x64	50x50	32x32
128x128	100%	100%	77%	70%	32%
91x91	100%	100%	76%	68%	33%
64x64	94%	94%	100%	91%	67%
50x50	79%	78%	98%	100%	86%
32x32	65%	67%	80%	92%	100%

Table: Performance portability on GeForce GTX1070 for different samples.

3D Fourier Reconstruction

	best runtime	tuning 50	tuning full
2080Ti	1m40s	88% $\pm$ 3%	54%
1070	5m49s	96% $\pm$ 2%	79%
750	16m59s	92% $\pm$ 4%	72%
680	15m12s	94% $\pm$ 2%	75%

Table: The relative performance of dynamically-tuned 3D Fourier reconstruction.