Autotuning

Introduction to autotuning, overview of our research

Jiří Filipovič et al.
Institute of Computer Science
Masaryk University

25. října 2018

Program development workflow

Implementation questions

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

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...

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.

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
- during autotuning, a combination of tuning parameters is repeatedly selected and empirically evaluated
- a search method is used to traverse the space of tuning parameters efficiently
- performed according to some objective, usually performance, but may be also energy consumption, numerical precision of pareto-optimal combination of several objectives

Taxonomy of Autotuning

Tuning scope

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

Tuning time

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

Developer involvement

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

Our focus

We target autotuning of code 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 online 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/4 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];
}
```

Is autotuning worthwile?

OK, so there are multiple variants of a code, but does it make sense to autotune?

- yes, tuning parameters interact, some sort of automatic search make sense

And wouldn't be enough to use a simple script?

- let's consider 3D Fourier Reconstruction[1] as an example
- the complex code in CUDA, brings an order of magnitude speedup over parallel CPU implementation
- we have identified 7 tuning parameters forming a tuning space of 430 configurations
- we have tuned it for different GPUs to see performance portability

[1] D. Střelák, C. O. S. Sorzano, J. M. Carazo, J. Filipovič. A GPU Acceleration of 3D Fourier Reconstruction in Cryo-EM, accepted in International Journal of High Performance Computing Applications.

3D Fourier Reconstruction Portability

Tabulka : Performance portability of 3D Fourier Reconstruction

	P100	GTX1070	GTX750	GTX680
Tesla P100	100%	95%	44%	96%
GTX 1070	88%	100%	31%	50%
GTX 750	65%	67%	100%	94%
GTX 680	71%	72%	71%	100%

We can gain over 3× speedup when tuning for each GPU architecture.

3D Fourier Reconstruction Portability

Tabulka : Sensitivity on input images in 3D Fourier Reconstruction (GTX 1070)

	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%

We can gain over $3\times$ speedup when tuning for specific input size.

Is autotuning worthwile?

It is impractical to re-tune implementation for each combination of HW and input manually.

- even offline tuning is not practical here, as we have too much combinations
- the best solution is to tune application when HW and input size is defined

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 online 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

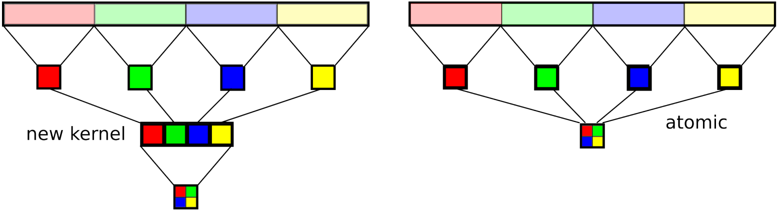
```
// Initialize tuner and kernel
ktt::Tuner tuner(platformIndex, deviceIndex);
const ktt::DimensionVector ndRangeDimensions(inputSize);
const ktt::DimensionVector workGroupDimensions(128);
ktt::KernelId foo = tuner.addKernelFromFile(kernelFile, "foo",
  ndRangeDimensions, workGroupDimensions);

// 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);
```

Unique features of KTT

Cross-kernel optimizations

- the user can add specific code for kernels execution
- 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



Unique features of KTT

Online autotuning

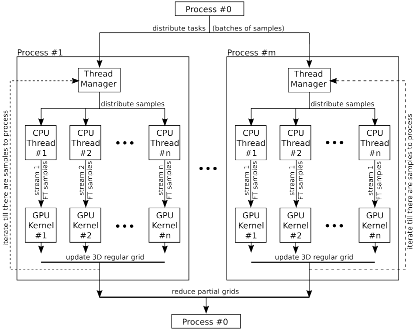
- KTT can be called to execute a kernel and retrieve results or try different combination of tuning parameters before the execution
- transparent for the application
- errors need to be handled explicitly
- tuning can be queried in any time

Online Tuning Sample

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

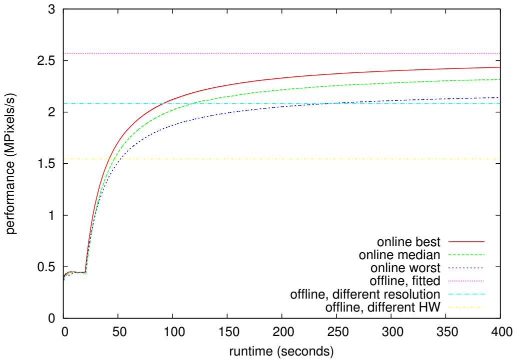
3D Fourier Reconstruction

Online tuning must mimic rich functionality of OpenCL/CUDA API.



Obrázek : Architecture of 3D Fourier Reconstruction.

3D Fourier Reconstruction



Obrázek : Performance of online tuned 3D Fourier reconstruction.