BOAST

Performance Portability Using Meta-Programming and Auto-Tuning

Frédéric Desprez [1], Brice Videau [1,3], Kevin Pouget [1], Luigi Genovese [2], Thierry Deutsch [2], Dimitri Komatitsch [3], Jean-François Méhaut [1]

[1]INRIA/LIG - CORSE, [2]CEA - L_Sim, [3]CNRS

Workshop CCDSC
October 6, 2016

Scientific Application Portability

Limited Portability

- Huge codes (more than 100 000 lines), Written in FORTRAN or C++
- Collaborative efforts
- Use many different programming paradigms (OpenMP, OpenCL, CUDA, ...)

But Based on Computing Kernels

- Well defined parts of a program
- Compute intensive
- Prime target for optimization

Kernels Should Be Written

- In a portable manner
- In a way that raises developer productivity
- To present good performance

HPC Architecture Evolution

Very Rapid and Diverse, Top500:

- Sunway processor (TaihuLight)
- Intel processor + Xeon Phi (Tianhe-2)
- AMD processor + nVidia GPU (Titan)
- IBM BlueGene/Q (Sequoia)
- Fujitsu SPARC64 (K Computer)
- Intel processor + nVidia GPU (Tianhe-1)
- AMD processor (Jaguar)

Tomorrow?

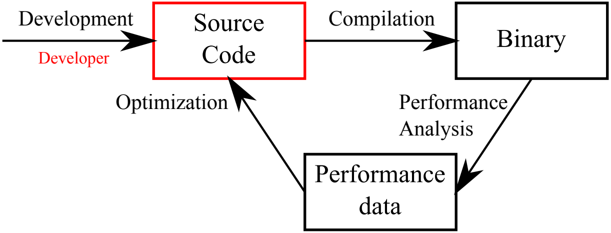
- ARM + DSP?
- Intel Atom + FPGA?
- Quantum computing?

How to write kernels that could adapt to those architectures?
(well maybe not quantum computing...)

Related Work

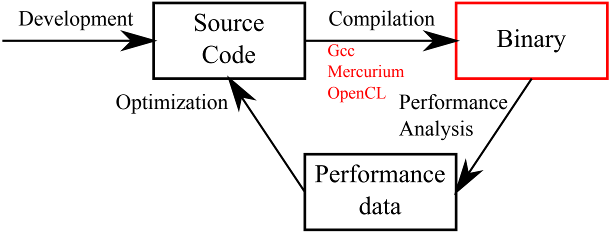
- **Ad hoc autotuners (usually for libraries):**
  - **Atlas** [6] (C macro processing)
  - **SPIRAL** [4] (DSL)
  - ...
- **Generic frameworks using annotation systems:**
  - **POET** [7] (external annotation file)
  - **Orio** [3] (source annotation)
  - **BEAST** [1] (Python preprocessor based, embedded DSL for optimization space definition/pruning)
- **Generic frameworks using embedded DSL:**
  - **Halide** [5] (C++, not very generic, 2D stencil targeted)
  - **Heterogeneous Programming Library** [2] (C++)

Classical Tuning of Computing Kernels



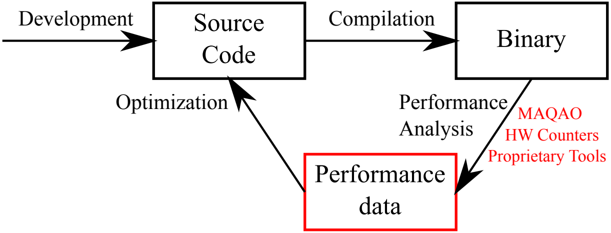
- Kernel optimization workflow
- Usually performed by a knowledgeable developer

Classical Tuning of Computing Kernels



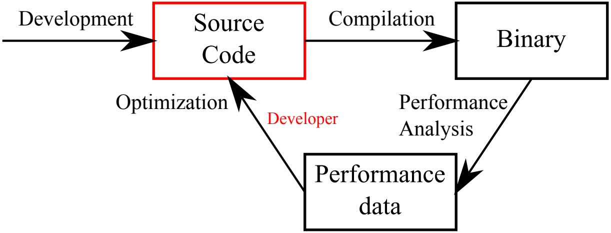
- Compilers perform optimizations
- Architecture specific or generic optimizations

Classical Tuning of Computing Kernels



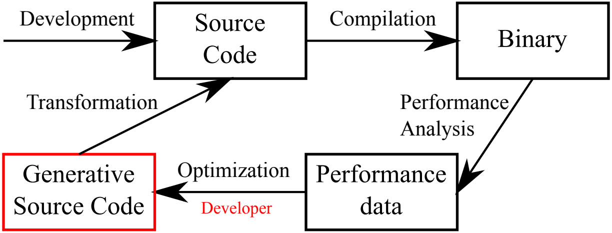
- Performance data hint at source transformations
- Architecture specific or generic hints

Classical Tuning of Computing Kernels



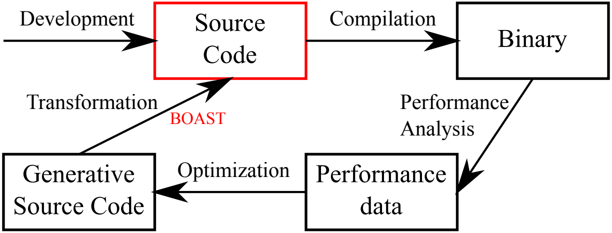
- Multiplication of kernel versions and/or loss of versions
- Difficulty to benchmark versions against each-other

BOAST Workflow



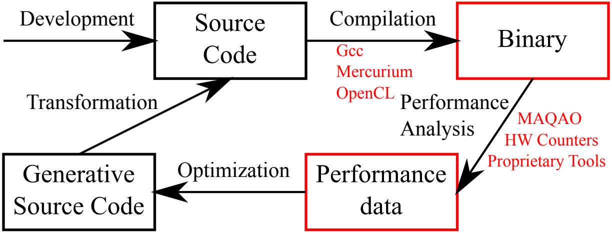
- Meta-programming of optimizations in BOAST
- High level object oriented language

BOAST Workflow



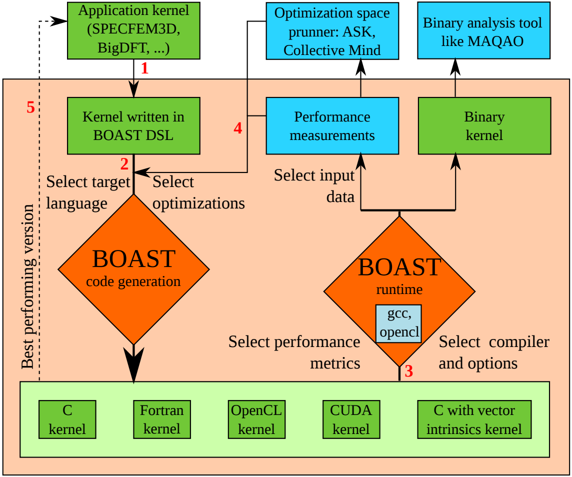
- Generate combination of optimizations
- C, OpenCL, FORTRAN and CUDA are supported

BOAST Workflow



- Compilation and analysis are automated
- Selection of best version can also be automated

BOAST Architecture



Example: Laplace Kernel from ARM

```
void laplace(const int width,
             const int height,
             const unsigned char src[height][width][3],
                   unsigned char dst[height][width][3]){
  for (int j = 1; j < height-1; j++) {
    for (int i = 1; i < width-1; i++) {
      for (int c = 0; c < 3; c++) {
        int tmp = -src[j-1][i-1][c] -   src[j-1][i][c] - src[j-1][i+1][c]\
                 - src[j  ][i-1][c] + 9*src[j  ][i][c] - src[j  ][i+1][c]\
                 - src[j+1][i-1][c] -   src[j+1][i][c] - src[j+1][i+1][c];
        dst[j][i][c] = (tmp < 0 ? 0 : (tmp > 255 ? 255 : tmp));
      }
    }
  }
}
```

- C reference implementation
- Many opportunities for improvement
- ARM GPU Mali 604 within the Montblanc project

Example: Laplace in OpenCL

```
kernel laplace(const int width,
               const int height,
               global const uchar *src,
               global       uchar *dst){
  int i = get_global_id(0);
  int j = get_global_id(1);
  for (int c = 0; c < 3; c++) {
    int tmp = -src[3*width*(j-1) + 3*(i-1) + c]\
              - src[3*width*(j-1) + 3*(i  ) + c]\
              - src[3*width*(j-1) + 3*(i+1) + c]\
              - src[3*width*(j  ) + 3*(i-1) + c]\
            + 9*src[3*width*(j  ) + 3*(i  ) + c]\
              - src[3*width*(j  ) + 3*(i+1) + c]\
              - src[3*width*(j+1) + 3*(i-1) + c]\
              - src[3*width*(j+1) + 3*(i  ) + c]\
              - src[3*width*(j+1) + 3*(i+1) + c];
    dst[3*width*j + 3*i + c] = clamp(tmp, 0, 255);
  }
}
```

- OpenCL reference implementation
- Outer loops mapped to threads
- 1 pixel per thread

Example: Vectorizing

```
kernel laplace(const int width,
               const int height,
               global const uchar *src,
               global       uchar *dst){
  int i = get_global_id(0);
  int j = get_global_id(1);
  uchar16 v11_ = vload16( 0, src + 3*width*(j-1) + 3*5*i - 3 );
  uchar16 v12_ = vload16( 0, src + 3*width*(j-1) + 3*5*i     );
  uchar16 v13_ = vload16( 0, src + 3*width*(j-1) + 3*5*i + 3 );
  uchar16 v21_ = vload16( 0, src + 3*width*(j  ) + 3*5*i - 3 );
  uchar16 v22_ = vload16( 0, src + 3*width*(j  ) + 3*5*i     );
  uchar16 v23_ = vload16( 0, src + 3*width*(j  ) + 3*5*i + 3 );
  uchar16 v31_ = vload16( 0, src + 3*width*(j+1) + 3*5*i - 3 );
  uchar16 v32_ = vload16( 0, src + 3*width*(j+1) + 3*5*i     );
  uchar16 v33_ = vload16( 0, src + 3*width*(j+1) + 3*5*i + 3 );
  int16 v11 = convert_int16(v11_);
  int16 v12 = convert_int16(v12_);
  int16 v13 = convert_int16(v13_);
  int16 v21 = convert_int16(v21_);
  int16 v22 = convert_int16(v22_);
  int16 v23 = convert_int16(v23_);
  int16 v31 = convert_int16(v31_);
  int16 v32 = convert_int16(v32_);
  int16 v33 = convert_int16(v33_);
  int16 res = v22 * (int)9 - v11 - v12 - v13 - v21 - v23 - v31 - v32 - v33;
        res = clamp(res, (int16)0, (int16)255);
  uchar16 res_ = convert_uchar16(res);
  vstore8(res_.s01234567, 0, dst + 3*width*j + 3*5*i);
  vstore8(res_.s89ab,     0, dst + 3*width*j + 3*5*i + 8);
  vstore8(res_.scd,       0, dst + 3*width*j + 3*5*i + 12);
  dst[3*width*j + 3*5*i + 14] = res_.se;
}
```

- Vectorized OpenCL implementation
- 5 pixels instead of one (15 components)

Example: Synthesizing Vectors

```
ucharl6 v11_ = vload16( 0, src + 3*width*(j-1) + 3*5*i - 3 );
ucharl6 v12_ = vload16( 0, src + 3*width*(j-1) + 3*5*i     );
ucharl6 v13_ = vload16( 0, src + 3*width*(j-1) + 3*5*i + 3 );
ucharl6 v21_ = vload16( 0, src + 3*width*(j  ) + 3*5*i - 3 );
ucharl6 v22_ = vload16( 0, src + 3*width*(j  ) + 3*5*i     );
ucharl6 v23_ = vload16( 0, src + 3*width*(j  ) + 3*5*i + 3 );
ucharl6 v31_ = vload16( 0, src + 3*width*(j+1) + 3*5*i - 3 );
ucharl6 v32_ = vload16( 0, src + 3*width*(j+1) + 3*5*i     );
ucharl6 v33_ = vload16( 0, src + 3*width*(j+1) + 3*5*i + 3 );
```

Becomes

```
ucharl6 v11_ = vload16( 0, src + 3*width*(j-1) + 3*5*i - 3 );
ucharl6 v13_ = vload16( 0, src + 3*width*(j-1) + 3*5*i + 3 );
ucharl6 v12_ = ucharl6( v11_.s3456789a, v13_.s56789abc );
ucharl6 v21_ = vload16( 0, src + 3*width*(j  ) + 3*5*i - 3 );
ucharl6 v23_ = vload16( 0, src + 3*width*(j  ) + 3*5*i + 3 );
ucharl6 v22_ = ucharl6( v21_.s3456789a, v23_.s56789abc );
ucharl6 v31_ = vload16( 0, src + 3*width*(j+1) + 3*5*i - 3 );
ucharl6 v33_ = vload16( 0, src + 3*width*(j+1) + 3*5*i + 3 );
ucharl6 v32_ = ucharl6( v31_.s3456789a, v33_.s56789abc );
```

- Reducing the number of loads since the vector are overlapping
- Synthesizing loads should save bandwidth
- Could be pushed further

Example: Reducing Variable Size

```
int16 v11 = convert_int16(v11_);
int16 v12 = convert_int16(v12_);
int16 v13 = convert_int16(v13_);
int16 v21 = convert_int16(v21_);
int16 v22 = convert_int16(v22_);
int16 v23 = convert_int16(v23_);
int16 v31 = convert_int16(v31_);
int16 v32 = convert_int16(v32_);
int16 v33 = convert_int16(v33_);
int16 res = v22 * (int)9 - v11 - v12 - v13 - v21 - v23 - v31 - v32 - v33;
      res = clamp(res, (int16)0, (int16)255);
```

Becomes

```
short16 v11 = convert_short16(v11_);
short16 v12 = convert_short16(v12_);
short16 v13 = convert_short16(v13_);
short16 v21 = convert_short16(v21_);
short16 v22 = convert_short16(v22_);
short16 v23 = convert_short16(v23_);
short16 v31 = convert_short16(v31_);
short16 v32 = convert_short16(v32_);
short16 v33 = convert_short16(v33_);
short16 res = v22 * (short)9 - v11 - v12 - v13 - v21 - v23 - v31 - v32 - v33;
        res = clamp(res, (short16)0, (short16)255);
```

- Using smaller intermediary types could save registers

Example: Optimization Summary

- Very complex process (several other optimizations could be applied)
- Intimate knowledge of the architecture required
- Numerous versions to be benchmarked
- Difficult to test combination of optimizations:
  - Vectorization,
  - Intermediary data type,
  - Number of pixels processed,
  - Synthesizing loads.
- Can we use BOAST to automate the process?

Example: Laplace Kernel with BOAST

- Based on components instead of pixel
- Use tiles rather than only sequence of elements
- Parameters used in the BOAST version:
  - x_component_number: a positive integer
  - y_component_number: a positive integer
  - vector_length: 1, 2, 4, 8 or 16
  - temporary_size: 2 or 4
  - synthesizing_loads: true or false

Example: ARM results

Image Size	Naive (s)	Best (s)	Acceleration	BOAST (s)	Acceleration
768 x 432	0.0107	0.00669	x1.6	0.000639	x16.7
2560 x 1600	0.0850	0.0137	x6.2	0.00687	x12.4
2048 x 2048	0.0865	0.0149	x5.8	0.00715	x12.1
5760 x 3240	0.382	0.0449	x8.5	0.0325	x11.8
7680 x 4320	0.680	0.0747	x9.1	0.0581	x11.7

- Optimal parameter values:
    - x_component_number: 16
    - y_component_number: 1
    - vector_length: 16
    - temporary_size: 2
    - synthesizing_loads: false
- Close to what ARM engineers found

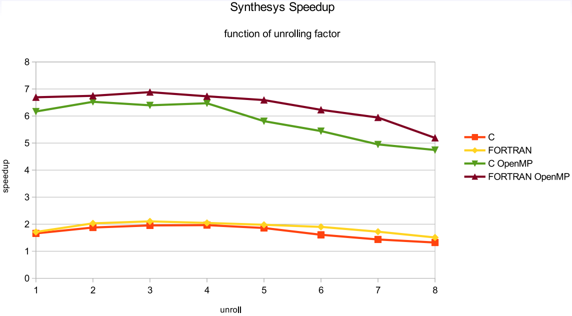
Example: Performance Portability

Image Size	BOAST ARM (s)	BOAST Intel	Ratio	BOAST NVIDIA	Ratio
768 x 432	0.000639	0.000222	x2.9	0.0000715	x8.9
2560 x 1600	0.00687	0.00222	x3.1	0.000782	x8.8
2048 x 2048	0.00715	0.00226	x3.2	0.000799	x8.9
5760 x 3240	0.0325	0.0108	x3.0	0.00351	x9.3
7680 x 4320	0.0581	0.0192	x3.0	0.00623	x9.3

- Optimal parameter values (Intel I7 2760, 2.4 GHz):
  - x_component_number: 16
  - y_component_number: 4..2
  - vector_length: 8
  - temporary_size: 2
  - synthesizing_loads: false
- Optimal parameter values nVidia (GTX 680):
  - x_component_number: 4
  - y_component_number: 4
  - vector_length: 4
  - temporary_size: 2
  - synthesizing_loads: false

Performance portability among several different architectures.

Real Applications: BigDFT



- Novel approach for DFT computation based on Daubechies wavelets
- Fortran and C code, MPI, OpenMP, supports CUDA and OpenCL
- Reference is hand tuned code on target architecture (Nehalem)
- Toward a BLAS-like library for wavelets

Real Applications: SPECFEM3D

- Seismic wave propagation simulator
- SPECFEM3D ported to OpenCL using BOAST
  - Unified code base (CUDA/OpenCL)
  - Refactoring: kernel code base reduced by 40%
  - Similar performance on NVIDIA Hardware
  - Non regression test for GPU kernels
- On the Mont-Blanc prototype:
  - OpenCL+MPI runs
  - Speedup of 3 for the GPU version

Conclusions

- BOAST v1.0 is released
- BOAST language features:
  - Unified C and FORTRAN with OpenMP support,
  - Unified OpenCL and CUDA support,
  - Support for vector programming.
- BOAST runtime features:
  - Generation of parametric kernels,
  - Parametric compilation,
  - Non-regression testing of kernels,
  - Benchmarking capabilities (PAPI support)

Perspectives

- Find and port new kernels to BOAST (GYSELA)
- Couple BOAST with other tools:
  - Parametric space pruners (speed up optimization),
  - Binary analysis (guide optimization, MAQAO),
  - Source to source transformation (improve optimization),
  - Binary transformation (improve optimization).
- Improve BOAST:
  - Improve the eDSL to make it more intuitive,
  - Better vector support,
  - Gather feedback.

Bibliography

Hartwig Anzt, Blake Haugen, Jakub Kurzak, Piotr Luszczek, and Jack Dongarra.
Experiences in autotuning matrix multiplication for energy minimization on gpus.
Concurrency and Computation: Practice and Experience, 27(17):5096–5113, 2015.
cpe.3516.

Jorge F. Fabeiro, Diego Andrade, and Basilio B. Fraguela.
Writing a performance-portable matrix multiplication.
Parallel Comput., 52(C):65–77, February 2016.

Albert Hartono, Boyana Norris, and Ponnuswamy Sadayappan.
Annotation-based empirical performance tuning using Orio.
In *Proceedings of the 23rd IEEE International Parallel & Distributed Processing Symposium*, Rome, Italy, 2009.
Also available as Preprint ANL/MCS-P1556-1008.

Markus Püschel, José MF Moura, Bryan Singer, Jianxin Xiong, Jeremy Johnson, David Padua, Manuela Veloso, and Robert W Johnson.
SPIRAL: A generator for platform-adapted libraries of signal processing algorithms.
International Journal of High Performance Computing Applications, 18(1):21–45, 2004.

Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe.
Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines.
ACM SIGPLAN Notices, 48(6):519–530, 2013.

R. Clint Whaley and Antoine Petitet.
Minimizing development and maintenance costs in supporting persistently optimized BLAS.
Software: Practice and Experience, 35(2):101–121, February 2005.

Qing Yi, Keith Seymour, Haihang You, Richard Vuduc, and Dan Quinlan.
POET: Parameterized optimizations for empirical tuning.
In *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*, pages 1–8. IEEE, 2007.