http://www.montblanc-project.eu

# BOAST

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

[1]CNRS, [2]CEA, [3]LIG

This project has received funding from the European Union's Seventh Framework Programme for research, technological development and demonstration under grant agreements n° 288777 and 610402.

# Scientific Application Portability

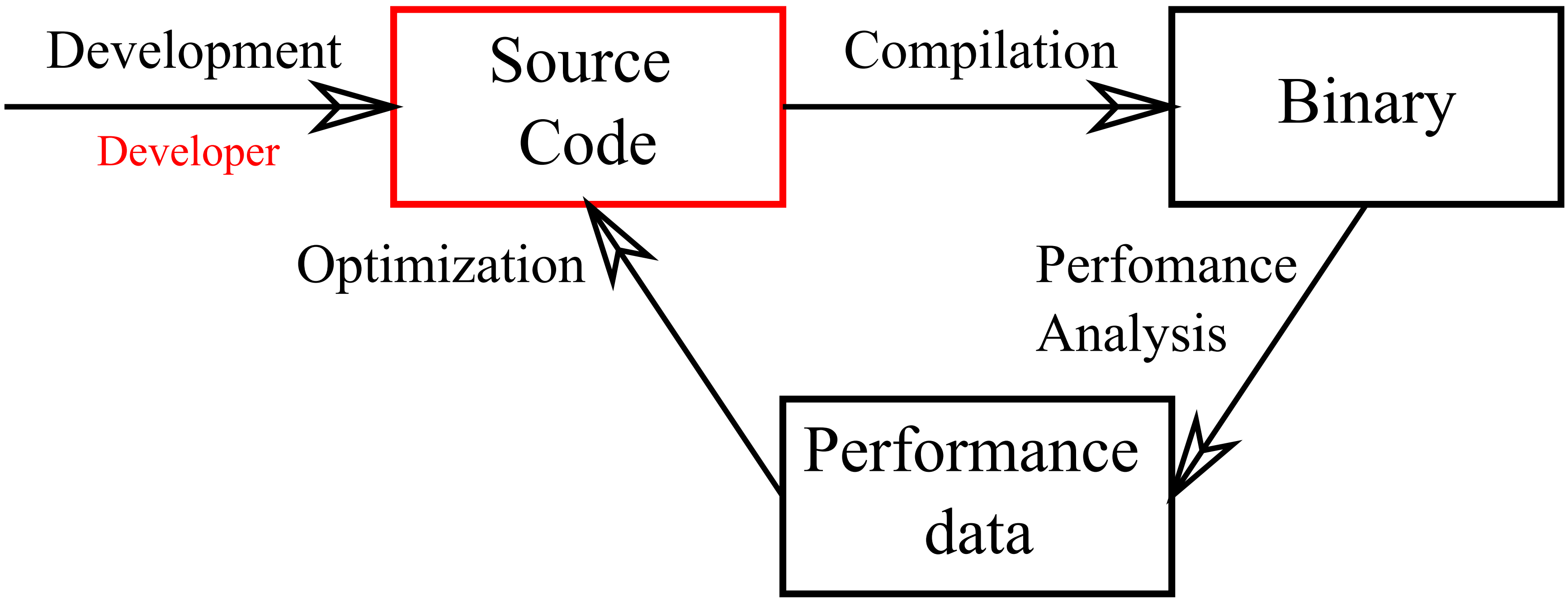
- Limited portability
  - Written in FORTRAN
  - Huge codes (more than ten thousands lines)
  - Collaborative efforts
  - Many programming paradigm can be used
- But based on *computing kernels*
  - Well defined part of a program
  - Compute intensive
  - Prime target for optimization
- Kernels should be:
  - Written in a *portable* manner
  - Written in a way that raises developer *productivity*
  - Written to present good *performance*

# HPC Architecture Evolution

- Very rapid and diverse, firsts from Top500:
  - Intel Processor + Xeon Phi (Tianhe-2)
  - AMD Processor + NVIDIA GPU (Titan)
  - IBM BlueGene/Q (Sequoia)
  - Fujitsu SPARC64 (K Computer)
  - Intel Processor + NVIDIA GPU (Thianhe-1)
  - AMD Processor (Jaguar)
- Tomorrow?
  - ARM + DSP?
  - Intel Atom + FPGA?
  - Quantum computing?
- How to write kernels that could adapt to those architectures?

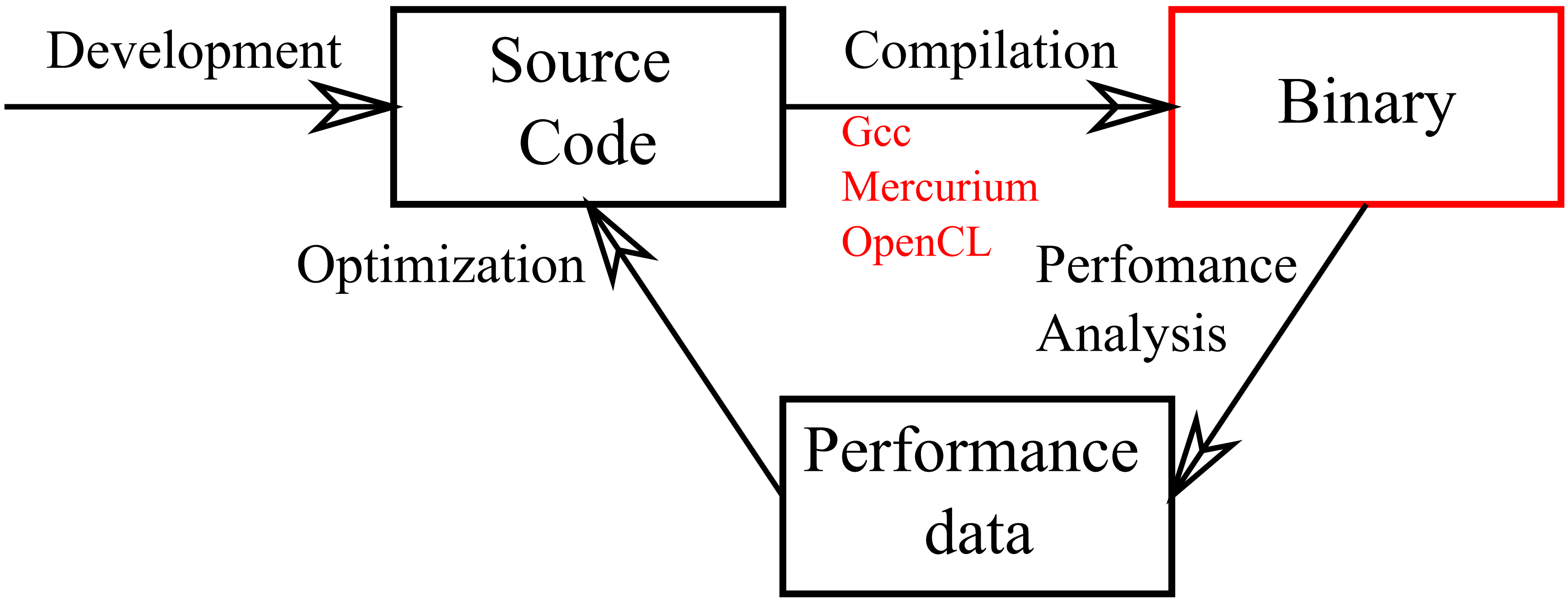
(well maybe not quantum computing)

# Classical Tuning of Computing Kernels



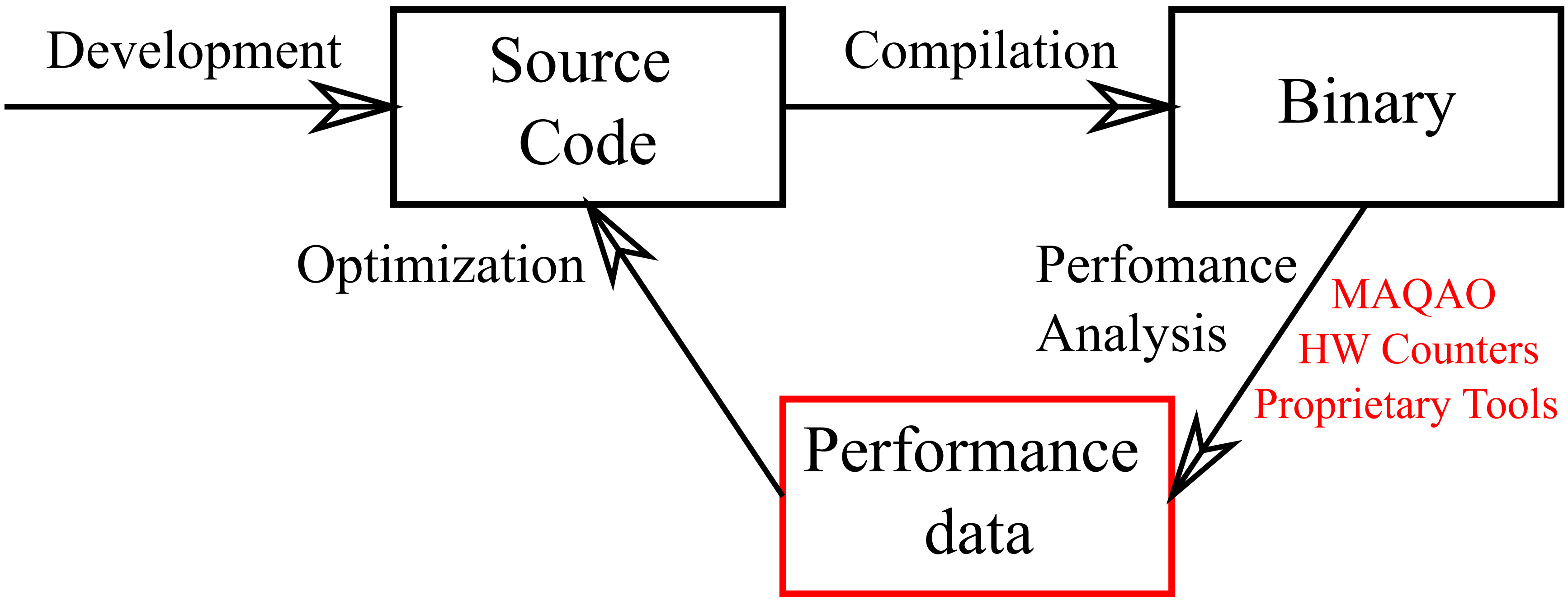

- Kernel Optimization Workflow
- Usually performed by a knowledgeable developer

# Classical Tuning of Computing Kernels



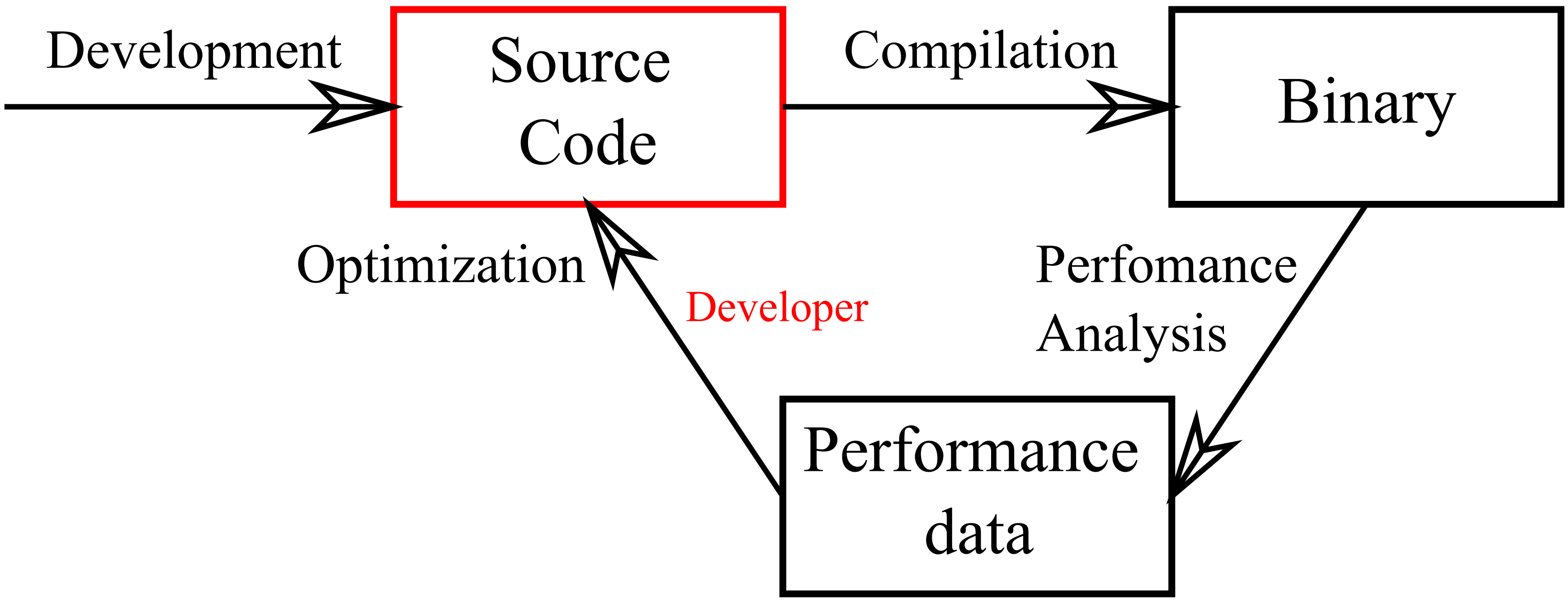

- Compilers perform optimizations
- Architecture specific of generic optimizations

# Classical Tuning of Computing Kernels



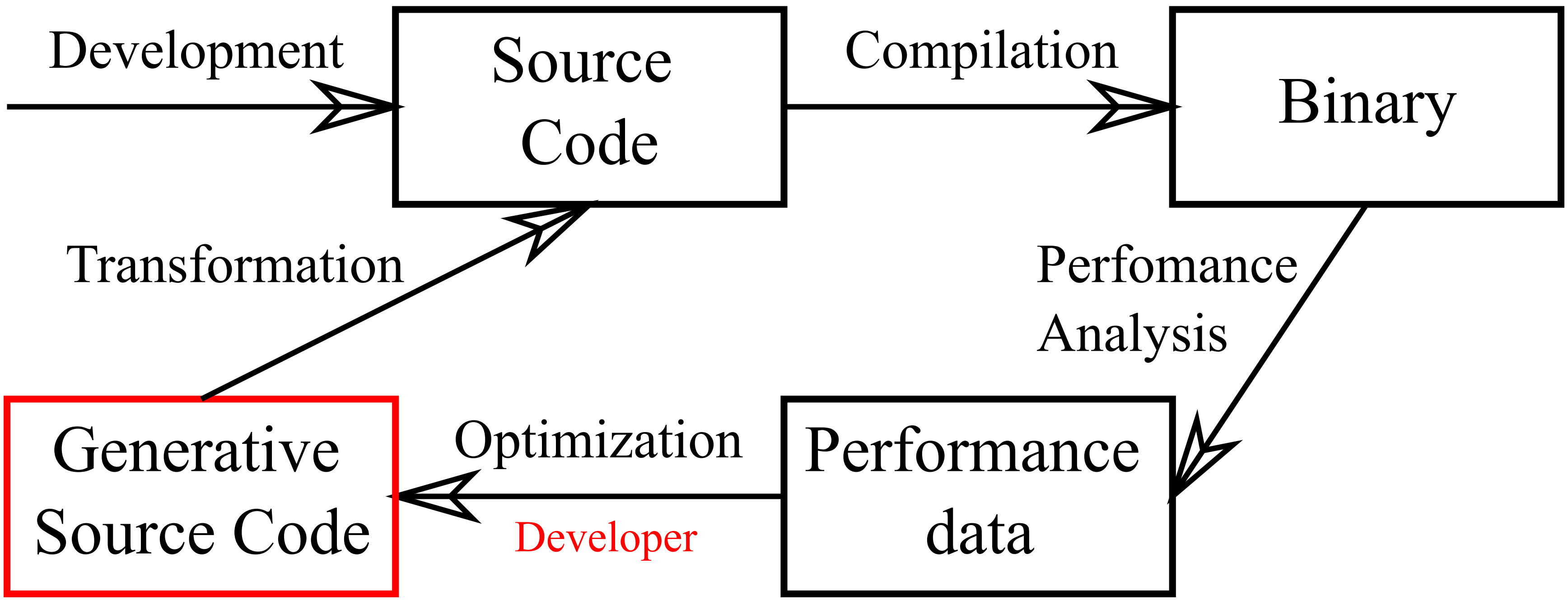

- Performance data hint at source transformations
- Architecture specific or generic hints

# Classical Tuning of Computing Kernels




- Multiplication of kernel versions or loss of versions
- Difficulty to Benchmark version against each other

# Tuning of Computing Kernels using BOAST

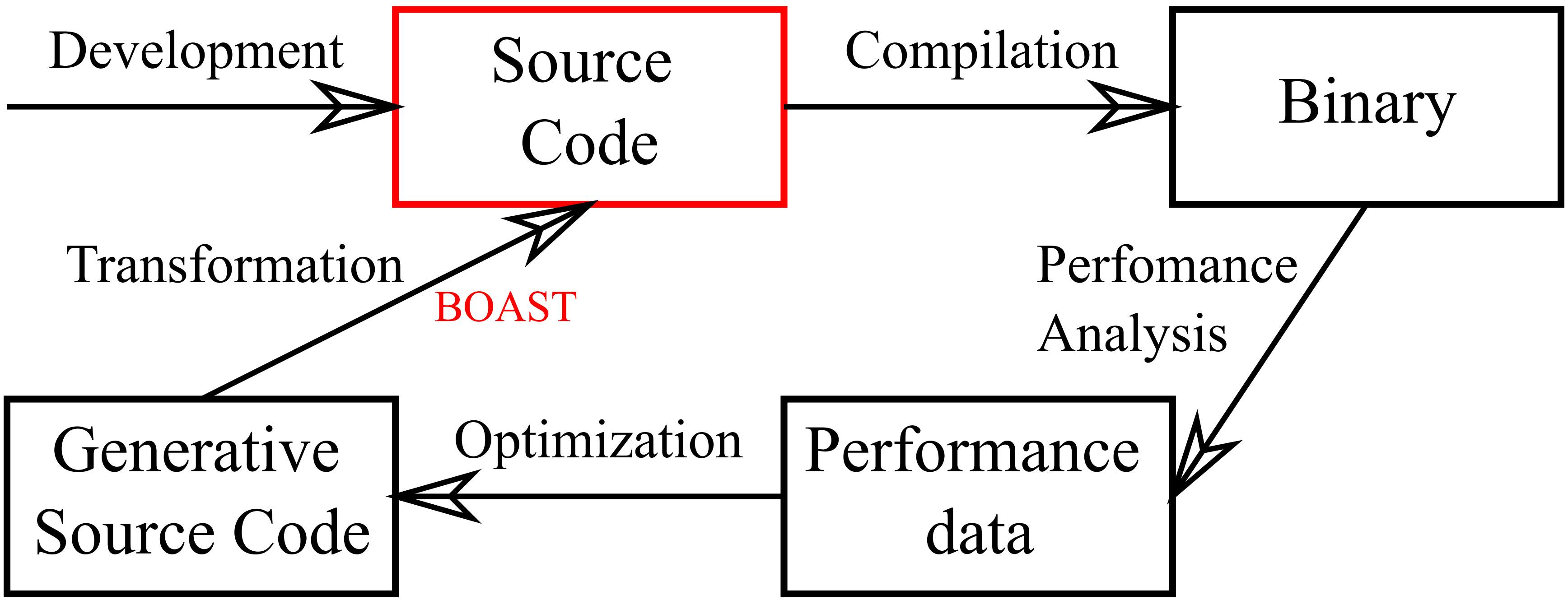



- Meta-programming of optimizations in BOAST
- High level object oriented language



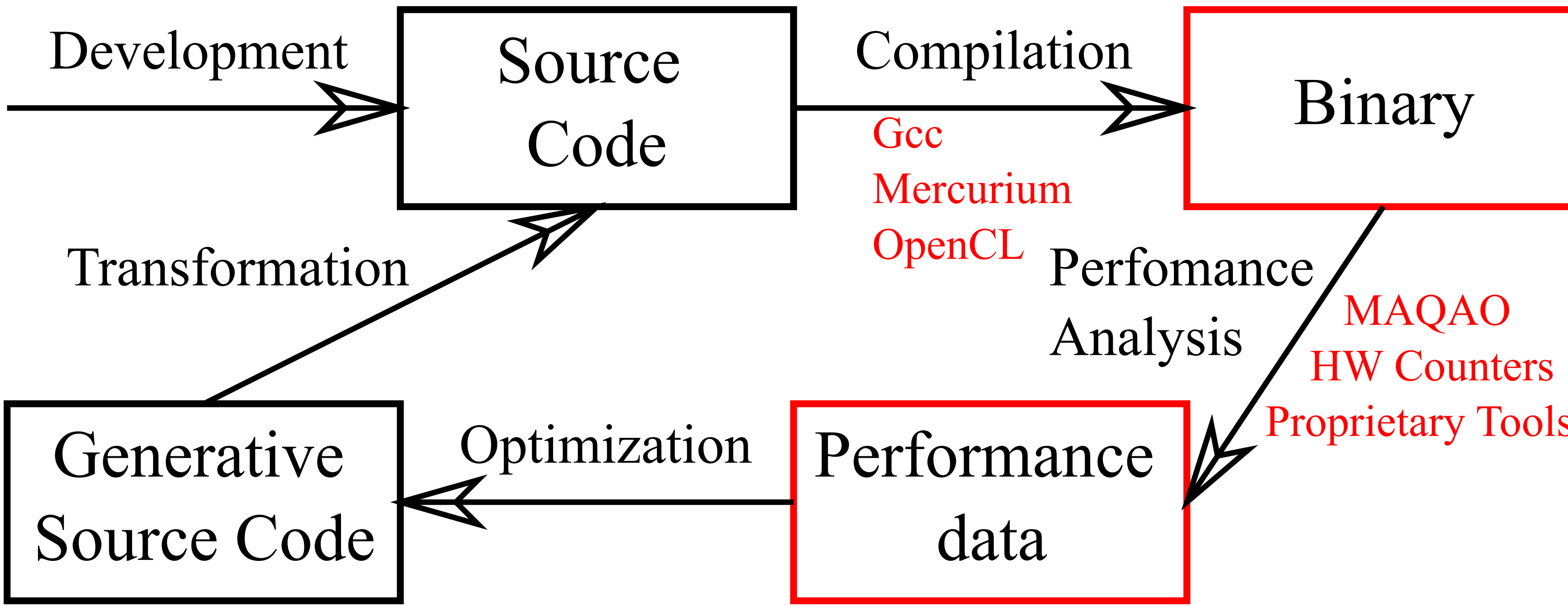
MONT-BLANC

# Tuning of Computing Kernels using BOAST



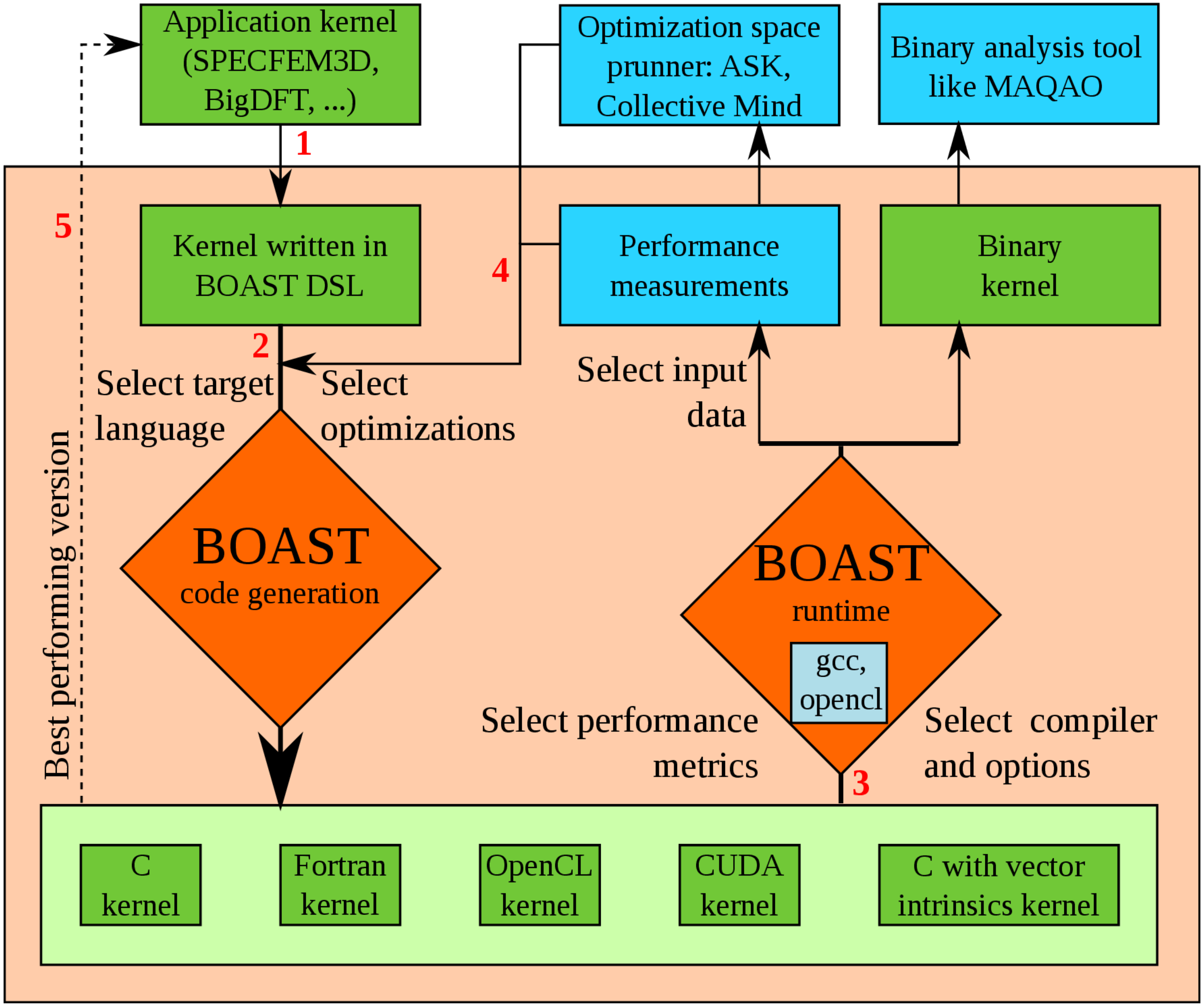

- Generate combination of optimizations
- C, OpenCL, CUDA, FORTRAN are supported

# Tuning of Computing Kernels using BOAST




- Compilation and analysis can be automated
- Select the best version for the target platform

# BOAST Architecture



# Example: Laplace Kernel from OpenCL Training

```
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

# Example: Laplace Kernel from OpenCL Training

```
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

# Example: Laplace Kernel from OpenCL Training

```
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);
  uchar 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 (15 components)

# Example: Laplace Kernel from OpenCL Training

```
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 );
```

Becomes

```
uchar16 v11_ = vload16( 0, src + 3*width*(j-1) + 3*5*i - 3 );
uchar16 v13_ = vload16( 0, src + 3*width*(j-1) + 3*5*i + 3 );
uchar16 v12_ = uchar16( v11_.s3456789a, v13_.s56789abc );
uchar16 v21_ = vload16( 0, src + 3*width*(j  ) + 3*5*i - 3 );
uchar16 v23_ = vload16( 0, src + 3*width*(j  ) + 3*5*i + 3 );
uchar16 v22_ = uchar16( v21_.s3456789a, v23_.s56789abc );
uchar16 v31_ = vload16( 0, src + 3*width*(j+1) + 3*5*i - 3 );
uchar16 v33_ = vload16( 0, src + 3*width*(j+1) + 3*5*i + 3 );
uchar16 v32_ = uchar16( v31_.s3456789a, v33_.s56789abc );
```

- Synthesizing loads should save bandwidth
- Could be pushed further

# Example: Laplace Kernel from OpenCL Training

```
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: Laplace Kernel from OpenCL Training

- Very complex process
- 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 rows
- 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
    - synthesize_loads: true or false

# Example: Laplace Kernel with BOAST : 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 |

Table 1: Best performance of ARM Laplace kernel.

- Optimal parameters values:
  - x_component_number = 16
  - y_component_number = 1
  - vector_length = 16
  - temporary_size = 2
  - synthesize_loads = false
- Close to what ARM engineers found

# Example: Laplace Kernel with BOAST : Results

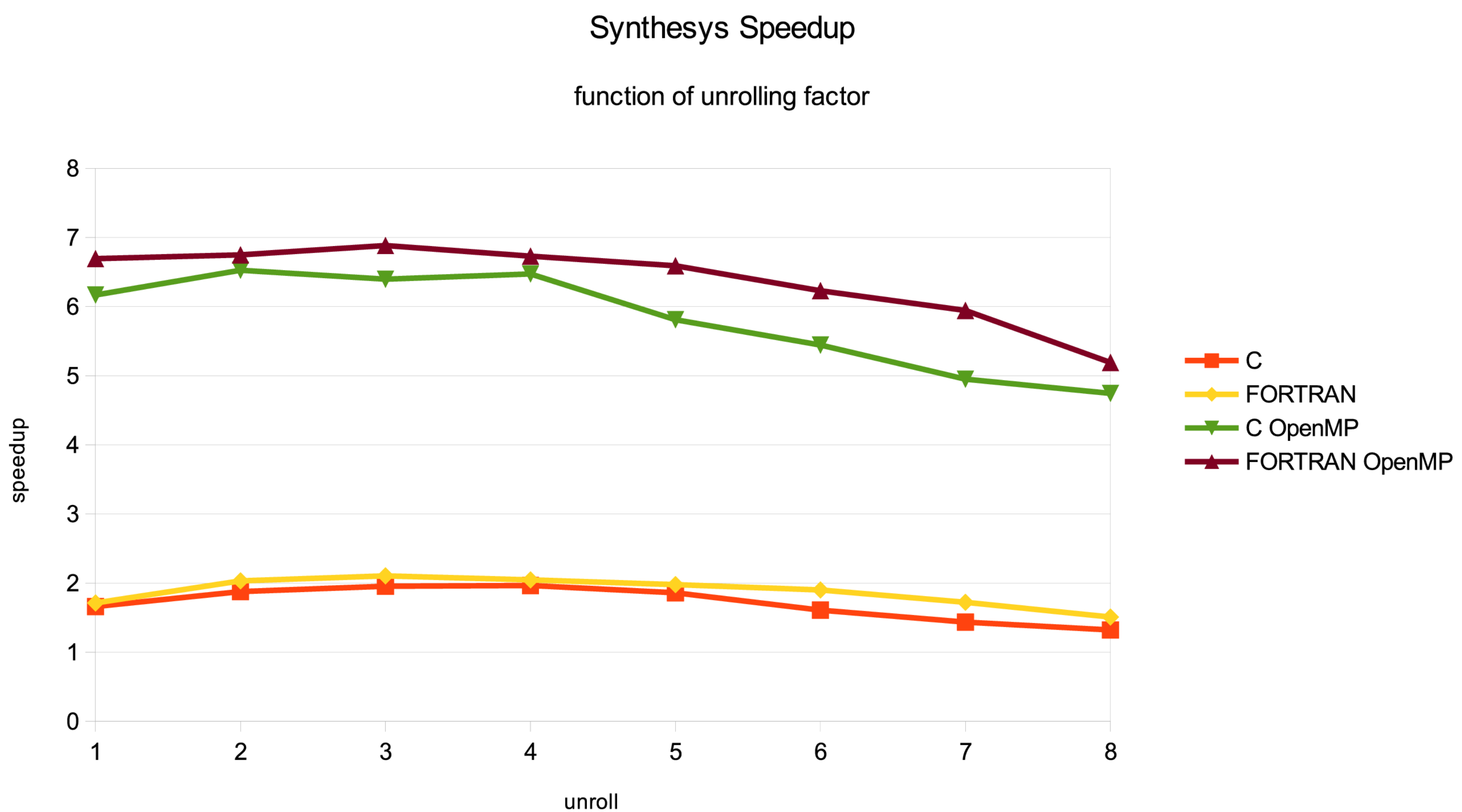
| Image Size | BOAST ARM (s) | BOAST Intel | Ratio | BOAST NVIDIA | Ratio |
|---|---|---|---|---|---|
| 768 x 432 | 0.000639 | 0.000222 | x2.9 | 0.0000715 | x8.9 |
| 2560 x 1600 | 0.00687 | 0.00222 | x3.1 | 0.000782 | x8.8 |
| 2048 x 2048 | 0.00715 | 0.00226 | x3.2 | 0.000799 | x8.9 |
| 5760 x 3240 |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

Table 1: Best performance of Laplace Kernel on several architectures.

- Optimal parameters values Intel:
  - x_component_number = 16
  - y_component_number = 4..2
  - vector_length = 8
  - temporary_size = 2
  - synthesize_loads = false
- Optimal parameters values NVIDIA:
  - x_component_number = 4
  - y_component_number = 4
  - vector_length = 4
  - temporary_size = 2
  - synthesize_loads = false

- Performance portability among several different architectures

# Real Applications: BigDFT




- Reference is hand tuned code on target architecture
- Toward a BLAS like library for wavelets

# Real Applications: SPECFEM3D

- SPECFEM3D ported to OpenCL using BOAST
  - Unified code base (CUDA/OpenCL)
  - Refactoring: kernel code base reduced by 40 percent
  - 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

# Conclusion

- BOAST version 1.0 is released with most of the envisioned features
- BOAST language features:
  - GPU support: unified OpenCL and CUDA
  - Unified C and FORTRAN with OpenMP support
  - Support for vector programming (in progress)
- BOAST runtime feature:
  - Generation of parametric kernels
  - Parametric compilation
  - Non-regression testing of kernels
  - Benchmarking capabilities (PAPI support)

# Future Work

- Find and port new kernels to BOAST, from other applications
- Continue improving the language:
  - Better vector support (by using higher level constructs)
  - Find other users and look at what they would like improved
- Improving BOAST runtime and modularity:
  - Use parametric space pruners like Adaptive Sampling Kit or Collective Mind to speed up optimization
  - Interface with MAQAO to guide optimization using binary analysis
  - Interface with binary or source to source optimizers