FROM RESEARCH TO INDUSTRY

Multi-Time Code Generation and Multi-Objective Code Optimization

PhD Defense

Victor LOMÜLLER

Thesis director: Henri-Pierre CHARLES

CEA, LIST

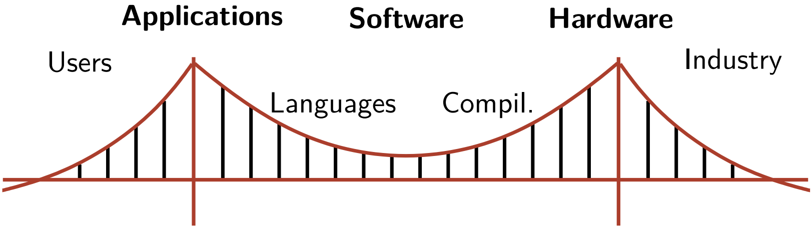
Laboratoire Infrastructure et Atelier Logiciel pour Puces

Grenoble – 12th November, 2014

www.cea.fr

leti & list

Semantic gap



Integer Computation
Floating Point Comput.
Telecommunication
Linear Algebra
Cryptography
2D/3D Graphic

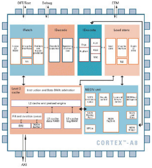
Floating Point Arith.
Integer Arith.

Floating Point Arith.
Integer Arith.
Vector Arith.

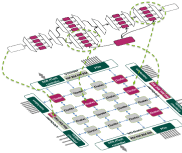
Floating Point Arith.
Integer Arith.
Complex Arith.
Saturated Arith.
Vector Arith.
Matrix Arith.
Bit Shuffling

The Landscape of Parallel Computing Research: A View from Berkeley

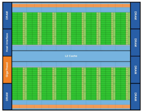
Architecture, Data and Performances



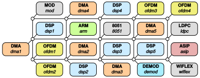
Cortex-A8



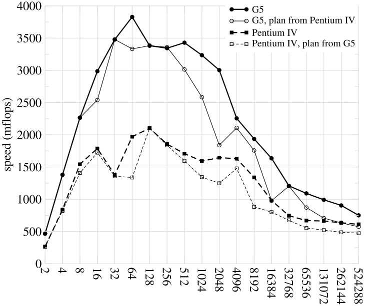
MPPA (Kalray)



CUDA Fermi



MAGALI (CEA)



Frigo et al. “The Design and Implementation of FFTW3”

→ Static tool-chains inadapted to use data and fit to the hardware

Thesis approach and contributions

Problematics

- What is data sensitiveness on massively parallel architectures
- How can it be used to improve performances ?
- What are the benefits and cost of dynamic code generation ?

Contribution

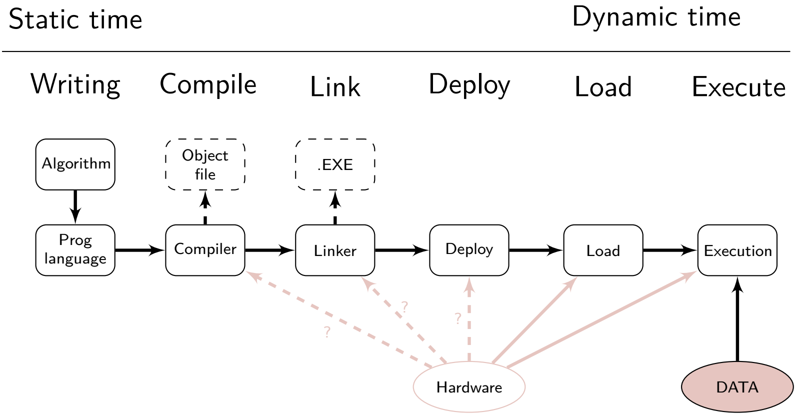
1. Adaptive library construction for GPU
   - Tuning with data sensitiveness in mind
   - Adaption for GPU
   - Code size control via runtime code generation
2. Code generation impact study
   - Complementary approach to runtime code generation
   - Execution speed, memory footprint and energy consumption study

Outline

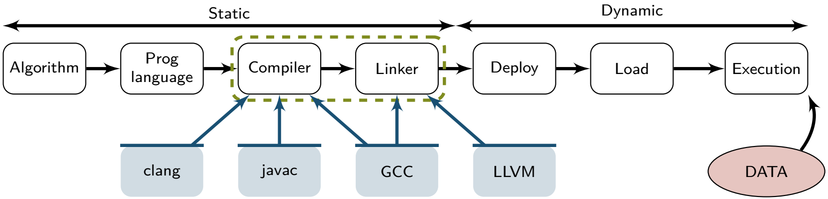
Outline

1. State of the Art
2. Adaptive library construction for GPU
3. Code generation impact study
4. Conclusion & Future

Compilation Times



Static Compilation



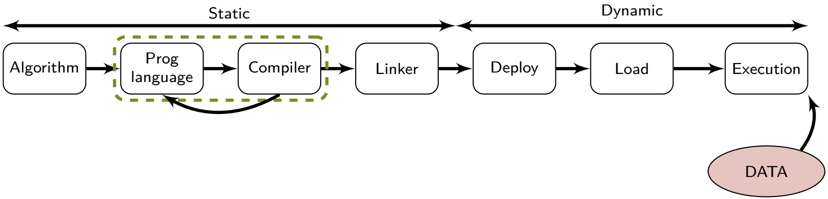
clang[14]/GCC[8]/LLVM[13]

- High-level language to binary object
- Link Time Optimization

javac[11]/LLVM[13]

- High-level language to bytecode
- Requires further operation

Source to Source



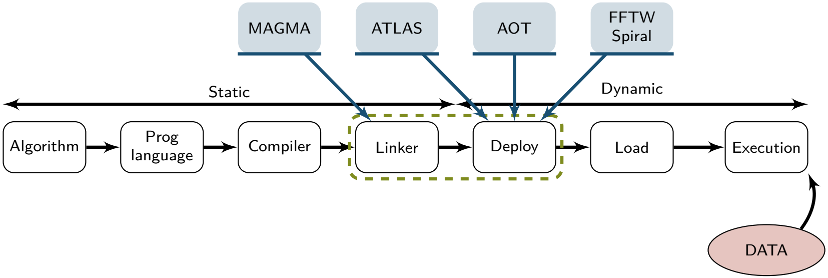
Spiral[18] & FFTW[6]

- Domain Specific Language
- Compiler uses domain specific knowledge
- Output C code

HMPP/OpenACC

- Code annotation
- Parallelizing code for massively parallel architecture

Deploy



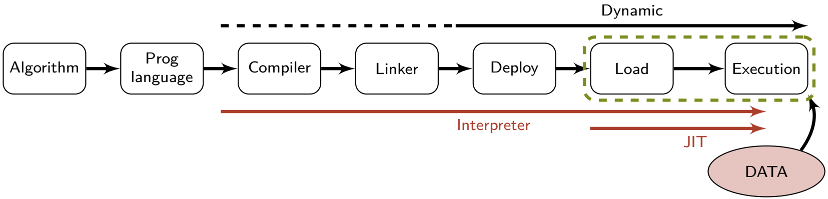
AOT: Ahead Of Time

- Compile bytecode to native instruction
- Android "*ART*" [2]

Tuning

- Set algorithm inner parameters to improve performances [6, 12, 18, 21]

Execution: Portability



Interpreter

Source Code to execution

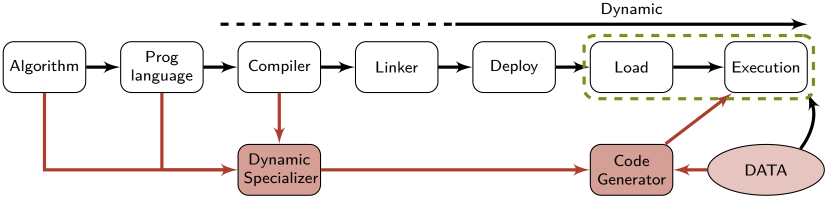
- Python [19]
- Perl [20]

Just-In-Time

Bytecode to execution

- Java/Dalvik Virtual Machine [11, 9]
- VMKit[7]

Execution: Performances



Implicit

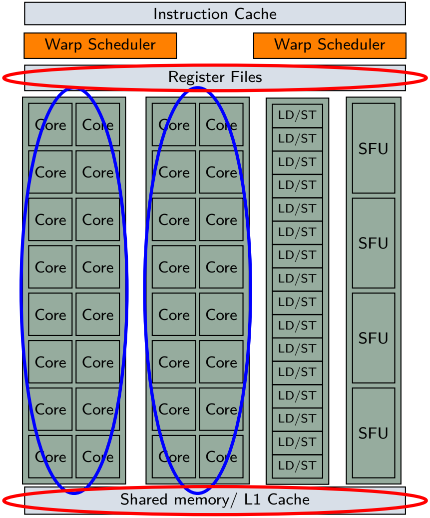
- IR reconstruction
  - Dynamo [1]
- Embedded IR
  - *Nuzman et al.* [16]

Explicit

- Code annotation
  - DyC [10], Tempo [4]
- Code generator expression
  - ‘C [17], HPBCG [3]

1. State of the Art
2. Adaptive library construction for GPU
3. Code generation impact study
4. Conclusion & Future

Graphic Processing Units



Used Platform

- *bullx* servers
- M2050 Fermi GPU
  - Compute capability 2.0
  - Single precision: 1,03 TFlops
  - Double precision: 515 GFlops

GEMM Tuning approach

GEMM

Matrix multiplication: $C = \alpha A \times B + \beta C$

- A: $M \times K$
- B: $K \times N$
- C: $M \times N$

GEMM tuning on GPU

- Set the number of thread per block (TB)
- Set the workload per thread: Instruction Level Parallelism (ILP)
- Set loading patterns (LP)

Algorithm tuning

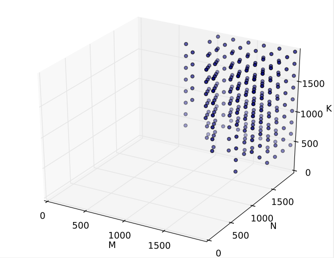
- Done without taking data shape into account
- Configuration is immutable

MAGMA Tuning and Limitation

- 32-bits floats
- $M \times K$ and $K \times N$ matrices

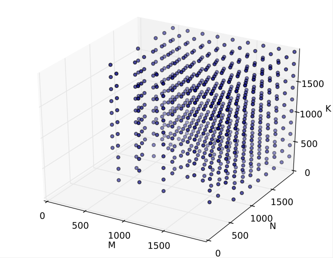
MAGMA

Perf. $>$ 573 GFlops



Ideal tuning

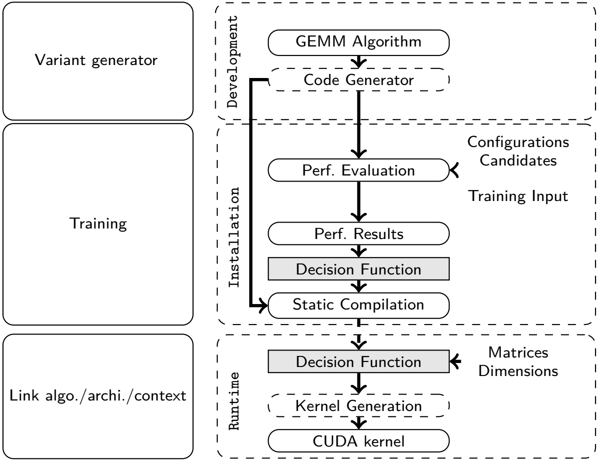
Perf. $>$ 573 GFlops



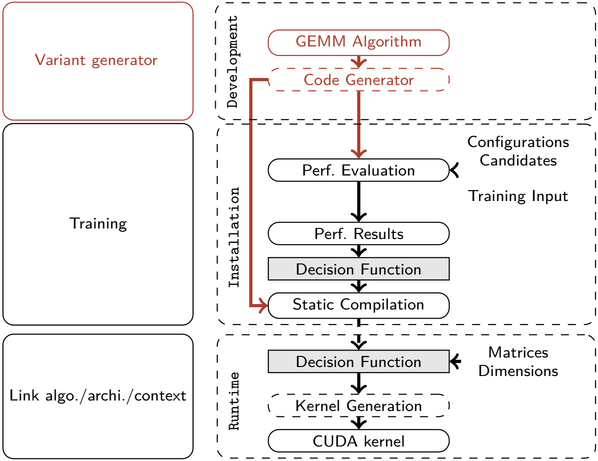
Objectives

- Use the best configuration to improve performances
- Keep library size under control

Adaptive Library Construction



Adaptive Library Construction

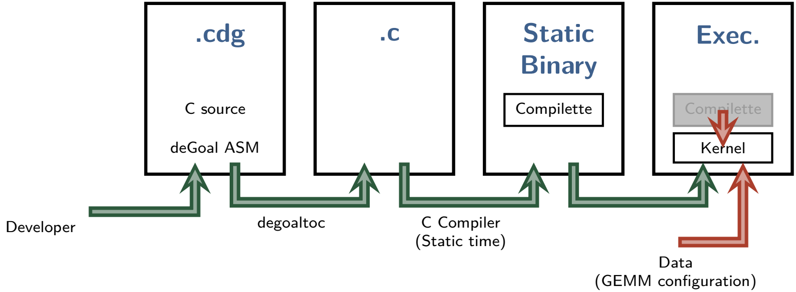


- Generate arbitrary GEMM implementation from a configuration, based on the algorithm from *Nath et al.* [15]
- Written in deGoal

deGoal

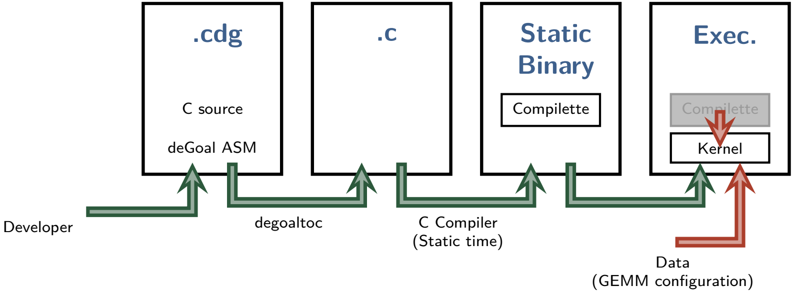
- Source to Source tool
- Builds runtime code generator and specializer called "*compilette*"
- For GPU target PTX IR code
- Requires to use the CUDA JIT

Variant Generator: deGoal



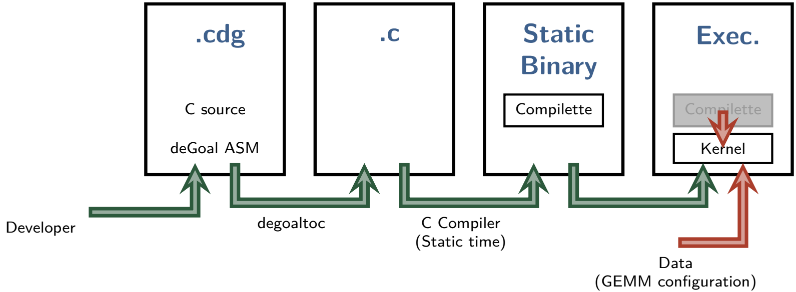

```
for (int j = 0; j < A_nbVect; j++) {
  #[ lw AtempWord(j), A, #(0), #(config->A_blockdim.x*
      alignementSize)
    add A, A, inputOffsetA
    sw asIdx, #(j*sharedAlignementSize*config->A_blockdim.y), #(
        AsLayout->Y*sharedAlignementSize*config->A_blockdim.x),
        AtempWord(j)
  ]#
}
```

Variant Generator: deGoal



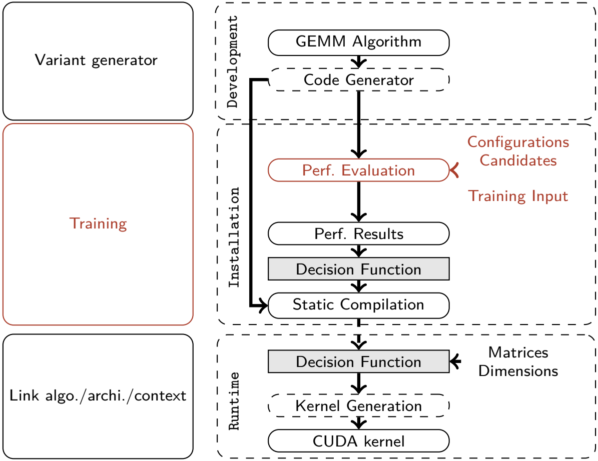

```
for (int j = 0; j < A_nbVect; j++) {
CDGgenlw_rrii(&(AtempWord[(j)]),&(A[0]),(0),(config->A_blockdim.x*
    alignementSize));
CDGgenadd_rrr(&(A[0]),&(A[0]),&(inputOffsetA[0]));
CDGgensw_riir(&(asIdx[0]),(j*sharedAlignementSize*config->
    A_blockdim.y),(AsLayout->Y*sharedAlignementSize*config->
    A_blockdim.x),&(AtempWord[(j)]));
}
```

Variant Generator: deGoal

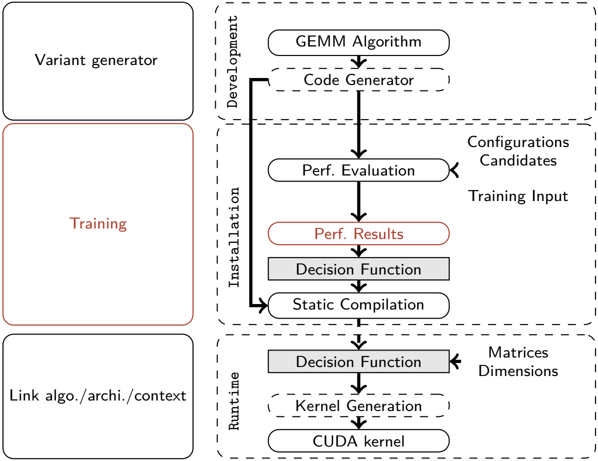



```
tex.1d.v4.s32.s32 {%tex_regA_tex0,%tex_regA_tex1,%tex_regA_tex2,%
    tex_regA_tex3},[A_tex,{%A0}];
mov.b32 %AtempWord0,%tex_regA_tex0;
add.s32 %A0,%A0,%inputOffsetA0;
st.shared.f32 [%asIdx0+8],%AtempWord0;
```

Adaptive Library Construction

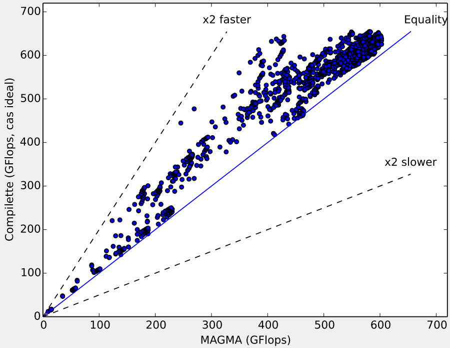


Adaptive Library Construction



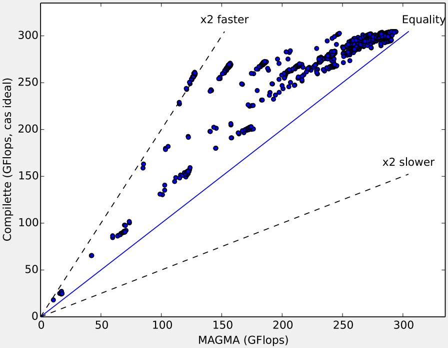
Space Exploration: Result

SGEMM



- 16 % gain in avg.
- 81 % max.

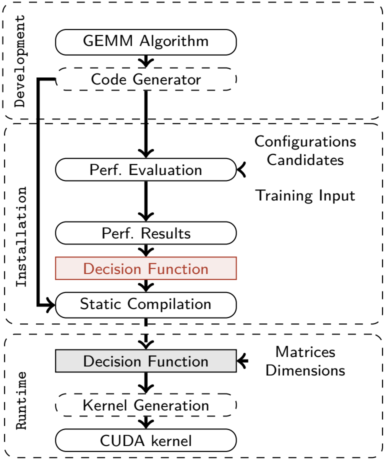
DGEMM



- 14 % gain in avg.
- 105 % max.

Adaptive Library Construction

Variant generator
Training
Link algo./archi./context



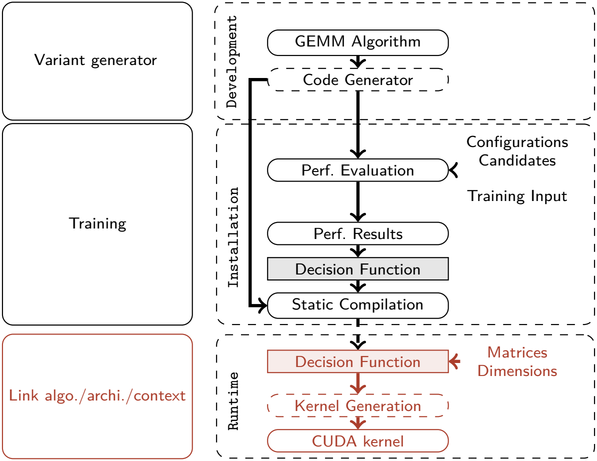
Inference Model Construction

How to chose the configuration to use according to matrix sizes ?

Machine Learning

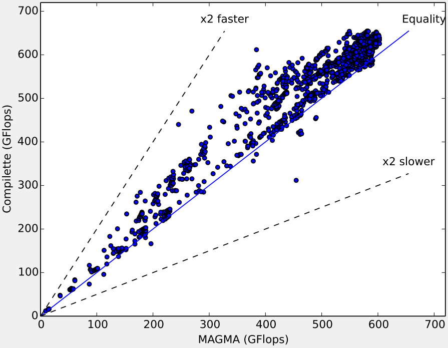
- Machine learning offer a way to infer the configuration from the results we obtained
- Many approaches exist
- We chose the C4.5 algorithm
  - Builds decision tree
  - Support for numerical features
  - Can be transformed into a small and fast decision function (set of rule)

Adaptive Library Construction

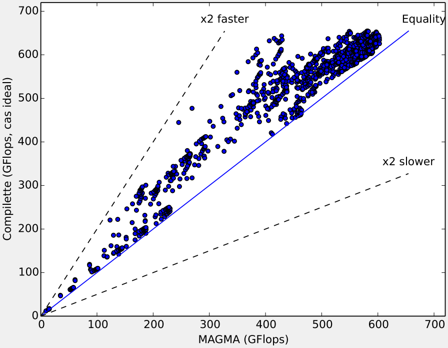


Results: Library Performances

SGEMM



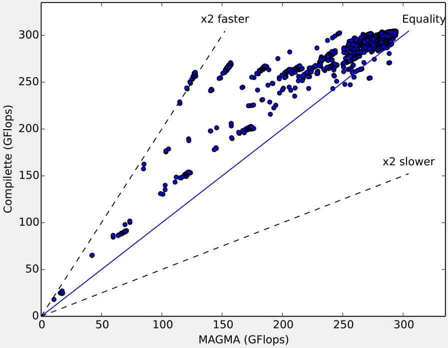
Ideal



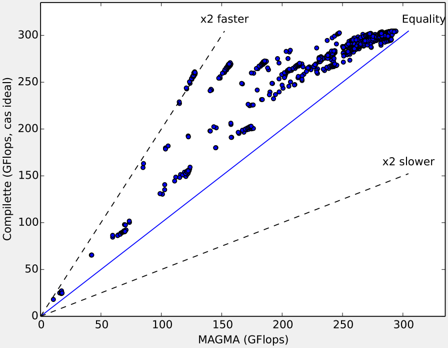
	Max. (%)	Average (%)	Configurations
Ideal	81	16	22
SGEMM	79	11	22

Results: Library Performances

DGEMM



Ideal



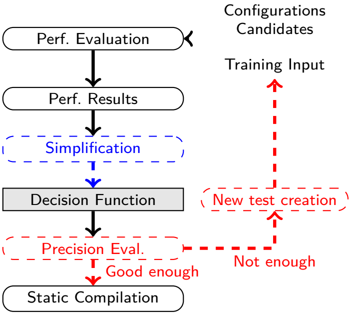
	Max. (%)	Average (%)	Configurations
Ideal	105	14	8
DGEMM	105	13	8

Conclusion

Result highlights

- Efficient library for the GEMM algorithm:
  - Speed up: 11 % for the SGEMM and 13 % for the DGEMM in average
  - Up to 105 %
- Library size reduced by a factor 40 compared to static specialization

Possible Evolution & Opening



Short term

- Simplification pass
- Precision evaluation and feedback loop
  - ASK [5]
- Test with other algorithms

Mid term

- Study for other architecture
  - Pulp
  - MPPA
- Data specialization of algorithms

Outline

Productivity ? Energy ? Memory ?

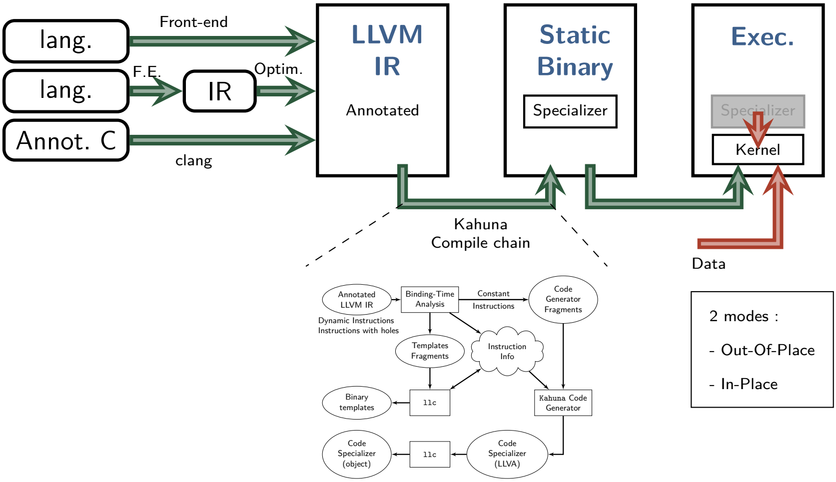
deGoal

- Flexible
- Fine grain optimization
- Efficient for small computation kernel
- Non adapted for large application

Specialization

- Improve execution speed
- What about energy/memory ?
- What about speed/energy/memory overhead ?

Kahuna: Automated Specializer Creation



2 modes :

- Out-Of-Place
- In-Place

Exp.: Audio Filter from SOX

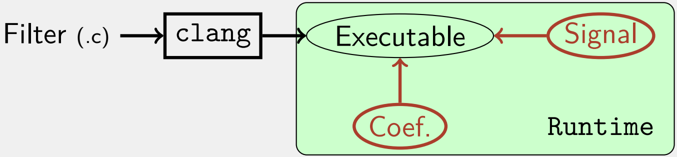
Input : in, out, $buffer\_length$, $filter$, $i1$, $i2$, $o1$, $o2$
for $l = 1$ **to** $buffer\_length$ **do**
  $sample \leftarrow$
  $in[l] \times filter.b0 + i1 \times filter.b1 + i2 \times filter.b2 - o1 \times filter.a1 - o2 \times filter.a2$
  $out[l] \leftarrow sample$
  $i2 \leftarrow i1$
  $i1 \leftarrow in[l]$
  $o2 \leftarrow o1$
  $o1 \leftarrow sample$
end
Return: $i1$, $i2$, $o1$, $o2$

Exp.: Audio Filter from SOX

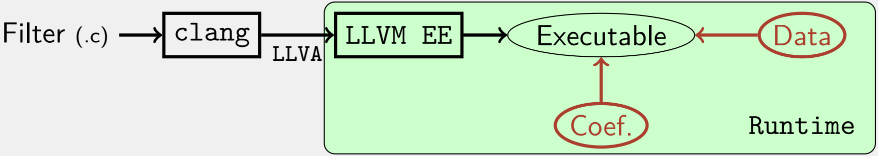
Input : in, out, $buffer\_length$, $filter$, $i1$, $i2$, $o1$, $o2$
for $l = 1$ **to** $buffer\_length$ **do**
  $sample \leftarrow$
  $in[l] \times filter.b0 + i1 \times filter.b1 + i2 \times filter.b2 - o1 \times filter.a1 - o2 \times filter.a2$
  $out[l] \leftarrow sample$
  $i2 \leftarrow i1$
  $i1 \leftarrow in[l]$
  $o2 \leftarrow o1$
  $o1 \leftarrow sample$
end
Return: $i1$, $i2$, $o1$, $o2$

Code Generation Models

Static binary

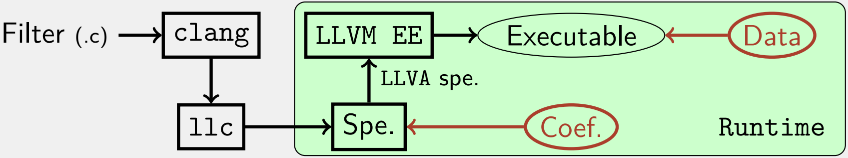


LLVM: JIT



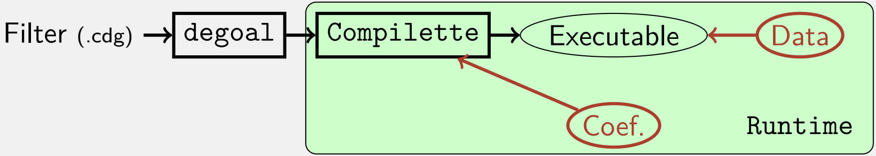
Code Generation Models

LLVM & Spe: JIT based specialization

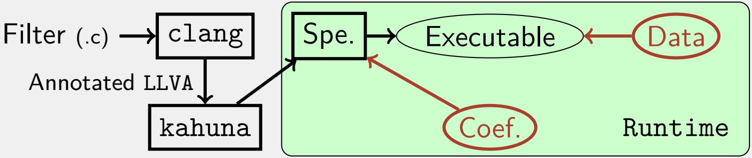


Code Generation Models

deGoal



Kahuna



Speed

- 800 MHz Cortex-A8 (Beagleboard-xM)
- Floating point operation made through the VFP3lite
- Theoretic peak: 80 MFlops

Energy

- Get using the simulator gem5 & McPAT, simulate the Cortex-A8
- Adapted to support in-order architecture (F. Endo works)

Memory

- Valgrind version 3.9 with the plugin *Massif*

Classical Results

		LLVM	LLVM & Spe	deGoal	Kahuna
Exec. Speed	Speed-up (%)	0	21	27	21
	Code gen. (cycles/insn)	3 M	3 M	233	20
	Recovery (samples)	-	4 M	285	10
	dev. time	-	30 min	1 week	5 min

- DyC, Tempo and `C equivalent to `deGoal` in terms of code generation speed

Less Common Results

		LLVM	LLVM & Spe	deGoal	Kahuna
Energy	Consumption	1	0.9	0.9	0.9
	Recovery (samples)	-	14 M	1 502	79
Memory	Static (Increase factor)	4852	4852	2.82	1.17
	Dyn. (heap/insn)	10 376	12 455	4	0
	Dyn. (stack)	26 312	22 744	2 984	2 984

Conclusion

- Automatic code generator creation from annotated IR using LLVM
- Lower development time for `Kahuna` with the C front-end

- Speed-up: 27 % for `deGoal`, 21 % for `Kahuna` and `LLVM & Spe O3`
- Very low recovery time for `deGoal` and `Kahuna`

- 10% energy reduction with `deGoal`, `Kahuna` and `LLVM & Spe O3`
- Very low recovery time with `deGoal` and `Kahuna`

- In-place strategy allows small footprints

Run-time optimization

- Strength reduction
- Dead code elimination
- Loop (partial) unrolling

Kahuna evolution

- deGoal interface
- Reinstantiation
- Incremental specialization

Future

- Implement run-time optimization
- Need to investigate more and bigger algorithms
- Integration into consequent tool chains
- Retargetable, incremental specialization

Outcomes

Adaptive library construction for GPU

- deGoal rewriting into a flexible architecture
- Retarget deGoal for GPU
- Data/algorithm sensitiveness study
- Library construction scheme to improve performances and controlling code size

Code generation impact study

- Development of Kahuna for automatic code specializer creation
- Study of the impact of code specialization on embedded device

Perspectives

Short term

- Adaptive library: learning improvement & algorithm extension
- Kahuna: run-time optimization & bigger kernels test

Mid term

- Adaptive library: architecture study & data adaptation
- Kahuna: reinstantiation

Long term

- Integration in virtual environment: seek performance portability
  $\rightarrow$ Need for prediction models
- Specialization strategy for constrained devices

Publications

Acted Conferences

- *Scilab on a Hybrid Platform*
  ParCo 2013, MS Heterogeneous Architectures; **V. Lomüller, S. Ledru, H.P. Charles**
- *A LLVM Extension for the Generation of Low Overhead Runtime Program Specializer*
  ADAPT 2014; **V. Lomüller, H.P. Charles**
- `deGoal` *a Tool to Embed Dynamic Code Generators Into Applications*
  Compiler Construction 2014; **H.P. Charles, D. Couroussé, V. Lomüller, F. A. Endo, R. Gauguey**

Non Acted Conferences

- *Progressive Compilation: New Metrics and Usages*
  Compilers for Parallel Computing 2013; **V. Lomüller, H.P. Charles**

Book Chapters

- *Data Size and Data Type Dynamic GPU Code Generation*
  GPU Design Patterns; **H.P. Charles, V. Lomüller**
- *Introduction to Dynamic Code Generation – a Simple Experiment with Matrix Multiplication for the STHORM Platform*
  Smart Multicore Embedded Systems; **D. Couroussé, H.P. Charles, V. Lomüller**

Poster

- *Maximizing GEMM Performances via Offline Heuristic Generation and Run-time Specialization*
  Advanced Computer Architecture and Compilation for High-Perfomance and Embedded Systems (ACACES 2013); **V. Lomüller, H.P. Charles**

Key Event

- *Exploiter pleinement les performances hardware multicœurs par compilation dynamique*
  V. Lomüller, H.P. Charles

In-progress

- *Low Overhead Code Specializations: A Case Study of the Impact on Speed, Energy and Memory*
  Compilers for Parallel Computing 2015; **V. Lomüller, H.P. Charles**

תודה Misaotra

감사합니다 ありがとう

teşekkürler Дякую

شكرا Merci Thank you

Mulțumesc Gracias

Dieureudieuf Obrigado

Mahalo Grazie

Bibliography I

Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia.
Dynamo: A Transparent Dynamic Optimization System.
In *Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation - PLDI '00*, 2000.

Brian Carlstrom, Anwar Ghuloum, and Ian Rogers.
The ART runtime.
Google I/O 2014, June 2014.

Henri-Pierre Charles and Khawar Sajjad.
HPBCG High Performance Binary Code Generator, 2009.

Charles Consel, Julia L. Lawall, and Anne-Françoise Le Meur.
A Tour of Tempo: a Program Specializer for the C Language.
Science of Computer Programming, 52(1-3):341–370, Jan 2004.

Bibliography II

Pablo de Oliveira Castro, Eric Petit, Jean Christophe Beyler, and William Jalby.
ASK: Adaptive Sampling Kit for Performance Characterization.
In Christos Kaklamanis, Theodore S. Papatheodorou, and Paul G. Spirakis, editors, *Euro-Par 2012 Parallel Processing - 18th International Conference*, volume 7484 of *Lecture Notes in Computer Science*, pages 89–101. Springer, 2012.

M. Frigo and S. G. Johnson.
The Design and Implementation of FFTW3.
Proceedings of the IEEE, 93(2):216–231, Feb 2005.

Bibliography III

Nicolas Geoffray, Gaël Thomas, Julia Lawall, Gilles Muller, and Bertil Folliot.
VMKit: A Substrate for Managed Runtime Environments.
In *Proceedings of the 6th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '10, pages 51–62, New York, NY, USA, 2010. ACM.

GNU Foundation.
GCC website.
https://gcc.gnu.org, 2014.

Google Corporation.
Dalvik Optimization and Verification, 2008.
https://android.googlesource.com/platform/dalvik/+/kitkat-release/docs/dexopt.html.

Bibliography IV

Brian Grant, Markus Mock, Matthai Philipose, Craig Chambers, and Susan J. Eggers.
The Benefits and Costs of DyC's Run-time Optimizations.
ACM Trans. Program. Lang. Syst., 22(5):932–972, September 2000.

Thomas Kotzmann, Christian Wimmer, Hanspeter Mössenböck, Thomas Rodriguez, Kenneth Russell, and David Cox.
Design of the Java HotSpot; Client Compiler for Java 6.
ACM Trans. Archit. Code Optim., 5(1):7:1–7:32, May 2008.

J. Kurzak, S. Tomov, and J. Dongarra.
Autotuning GEMM Kernels for the Fermi GPU.
Parallel and Distributed Systems, IEEE Transactions on, 23(11):2045 –2057, nov. 2012.

Bibliography V

Chris Lattner and Vikram Adve.
LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation.
In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, Mar 2004.

LLVM Foundation.
clang website.
`https://clang.llvm.org`, 2014.

Rajib Nath, Stanimire Tomov, and Jack Dongarra.
An Improved MAGMA GEMM For Fermi Graphics Processing Units.
Int. J. High Perform. Comput. Appl., 24(4):511–515, November 2010.

Dorit Nuzman, Revital Eres, Sergei Dyshel, Marcel Zalmanovici, and Jose Castanos.
JIT Technology with C/C++: Feedback-directed Dynamic Recompilation for Statically Compiled Languages.
ACM Trans. Archit. Code Optim., 10(4):59:1–59:25, December 2013.

Massimiliano Poletto, Wilson C. Hsieh, Dawson R. Engler, and M. Frans Kaashoek.
`C and tcc: A Language and Compiler for Dynamic Code Generation.
ACM Transactions on Programming Languages and Systems, 21(2):324–369, Jan 1999.

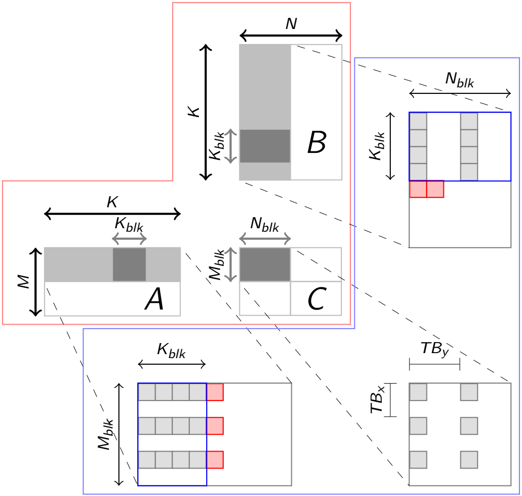
Markus Püschel, Franz Franchetti, and Yevgen Voronenko.
Encyclopedia of Parallel Computing, chapter Spiral.
Springer, 2011.

Python Software Foundation.
Python website.
https://www.python.org, 2014.

The Perl Foundation.
Perl website.
https://www.perl.org, 2014.

R. Clint Whaley and Jack Dongarra.
Automatically Tuned Linear Algebra Software.
In *SuperComputing 1998: High Performance Networking and Computing*, 1998.

GEMM Algorithm



Inference Model Construction

Features ?

- Matrix sizes: 3 dimensions (M, K, N)

Feature extension

- Add information such as number of elements, number of computation ...
- Not new information in itself but offer more possibilities for C4.5

Algorithm	Feature extension	Number of rules	Configurations
SGEMM	No	152	22
SGEMM	Yes	76	
DGEMM	No	121	8
DGEMM	Yes	84	

JIT Cost

Generation Time

- Average generation time: 58 ms
- Can be hidden through the use of a generic version

	SGEMM	DGEMM
Recovery with JIT	800	200
Recovery without (extrapolation)	16	2

Kahuna Build Process