Flexible and Performing Kernels Dynamically Generated with `deGoal` for Embedded Systems

Damien Couroussé(1) **Henri-Pierre Charles**(1) **Yves Lhuillier**(2)

author.name@cea.fr

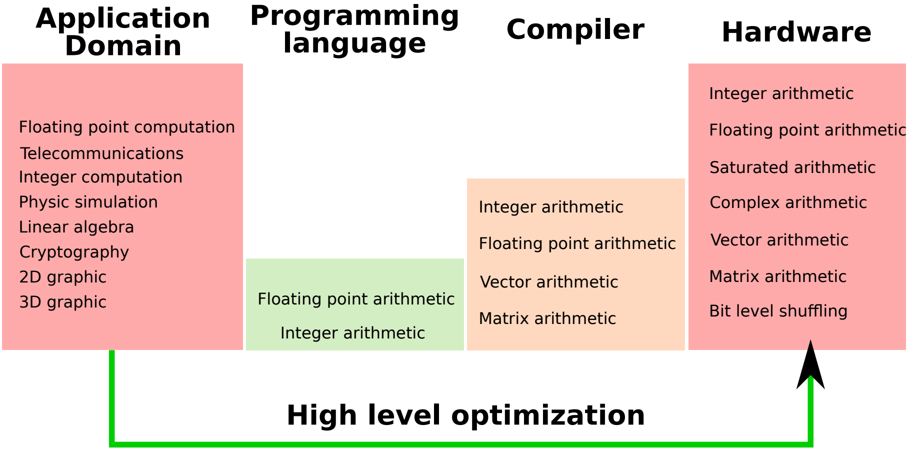
CEA-LIST, LaSTRE/Grenoble (1) and LCE (2) laboratories

December 1st, 2011

Outline

1. Overview of `deGoal`
   - motivation
   - a high level assembly language
   - the development workflow
2. Two application cases
   - Matrix multiplication
   - A memory allocator for embedded systems
3. Conclusion

the semantic bottleneck



motivation for developing `deGoal`

1. the semantic bottleneck of C language
2. limitations of static compilers
   - vector instructions: limited use
   - target's ISA is partially covered. eg: `psadd` (Cell), `fpx_MAC` (FPx on STxP70), etc.
   - **no code optimizations from execution context** (data values, memory addresses, etc.)
3. Claim: `C` source code for high performance is *not* portable
   - architecture-specific intrinsics
   - compiler-dependent pragmas
   - the final binary code depends on the compiler technology: `gcc`, `icc`, `open64`, `LLVM`, etc.

deGoal: Dynamic Execution of Generated Optimized kernel from Algorithmic Level

- code generation depends on the execution context
  - constant propagation from runtime data (values and memory addr.)
  - instruction selection
  - removal of dead code
- support of application-specific instructions
- a common language for all architectures
- suitable for many-core embedded systems
  - code generators have a very small footprint
  - very fast code generation (10 to 100 cycles / instruction)
  - supports heterogeneous architectures
  - control over code generation (not done behind developer's back)

Disclaimer

We focus at the processor's level (i.e. PEs in P2012).

a high level assembly language

Characteristics:

Architecture independent

Multiple arithmetics integer, floating point, complex, saturated

Register allocation virtual vector registers of multiple width,
automatic sizing of vectors (WIP)

Dynamically parametrized mix `C` expressions with generated code

Multimedia operators SAD, DFT, memory access patterns
operators are automatically adjusted to target capabilities
(e.g. availability of the `psadd` instruction)

No Intermediate Representation for very fast code generation

the development workflow

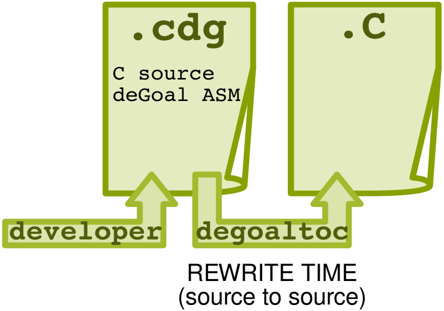
Compilette
: the runtime code generator,
kernel-specific
designed to be very fast

Kernel
: the code section to optimize,
generated at runtime

the development workflow

Compilette the runtime code generator, kernel-specific, designed to be very fast

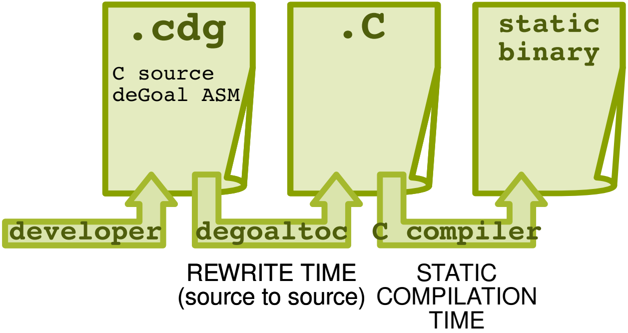
Kernel the code section to optimize, generated at runtime



the development workflow

Compilette the runtime code generator, kernel-specific designed to be very fast

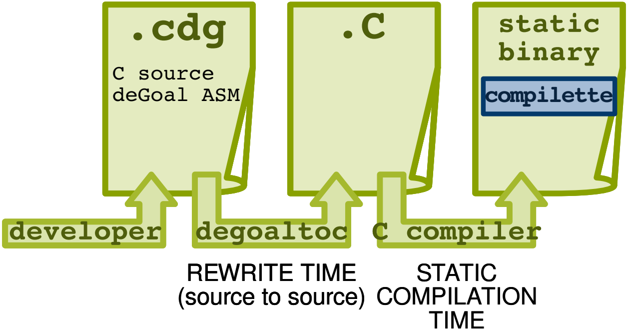
Kernel the code section to optimize, generated at runtime



the development workflow

Compilette the runtime code generator, kernel-specific designed to be very fast

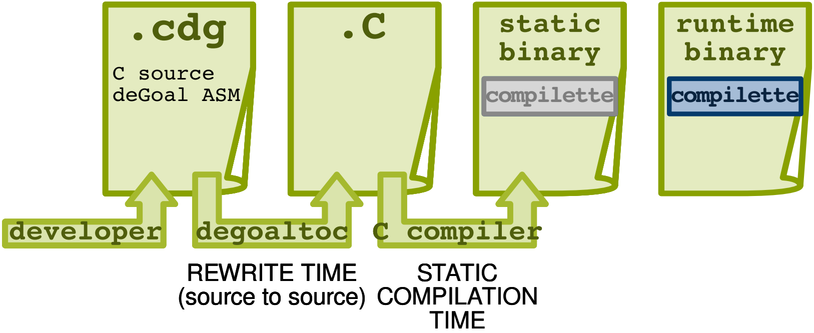
Kernel the code section to optimize, generated at runtime



the development workflow

Compilette the runtime code generator,
kernel-specific
designed to be very fast

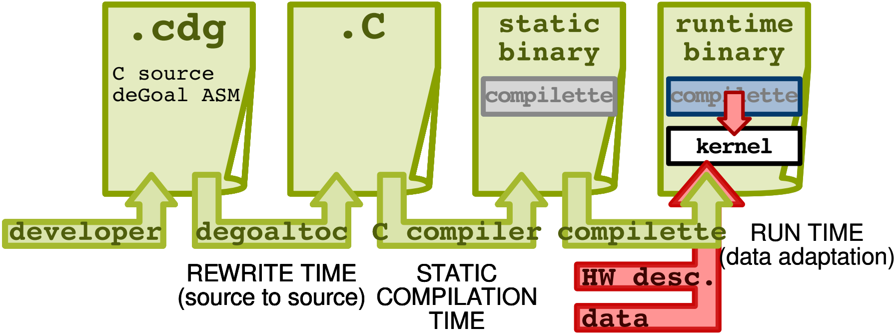
Kernel the code section to optimize,
generated at runtime



the development workflow

Compilette the runtime code generator, kernel-specific designed to be very fast

Kernel the code section to optimize, generated at runtime



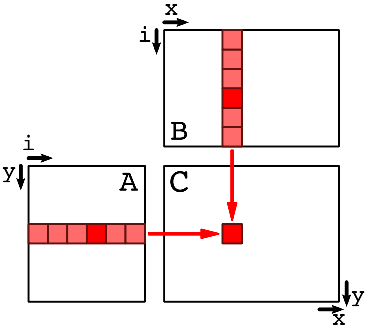
Outline

naive implementation

$$[C] = [A] \times [B] \quad (1)$$

$$\forall x \in \{0, \ldots, n-1\}, \forall y \in \{0, \ldots, q-1\}, c_{xy} = \sum_{i=0}^{p-1} a_{iy} b_{xi} \quad (2)$$

```
clear(C)
for (y=0; y < n; y++) {
  for (x=0; x < q; x++) {
    for (i=0; i < p; i++) {
      C[x,y] =
        C[x,y]+A[i,y]*B[x,i]
} } }
```



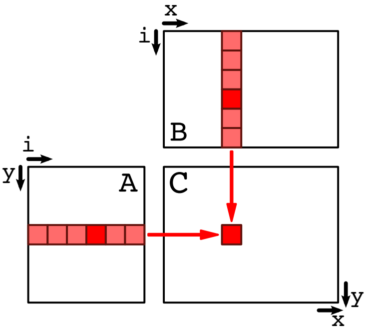
experiment #1 with `deGoal`

Objective: simple code generation, constant propagation only

- propagation of data constants into the binary code
- loop unrolling

```
// generate kernel
mac = gen_mac(&A, &B, &C);

clear(C)
for (y=0; y < n; y++){
  for (i=0; i < p; i+=VSIZE){
    mac(y,i);// execute kernel
} }
```



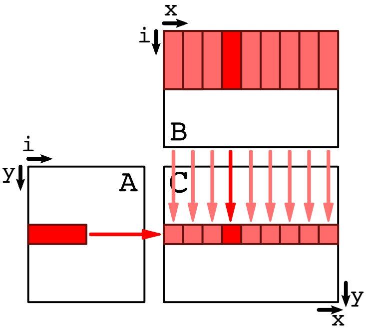
experiment #1 with `deGoal`

Objective: simple code generation, constant propagation only

- propagation of data constants into the binary code
- loop unrolling

```
// generate kernel
mac = gen_mac(&A, &B, &C);

clear(C)
for (y=0; y < n; y++){
  for (i=0; i < p; i+=VSIZE){
    mac(y,i);// execute kernel
} }
```



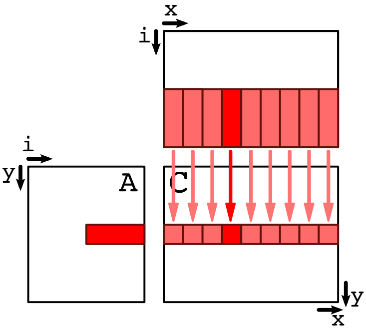
experiment #1 with deGoal

Objective: simple code generation, constant propagation only

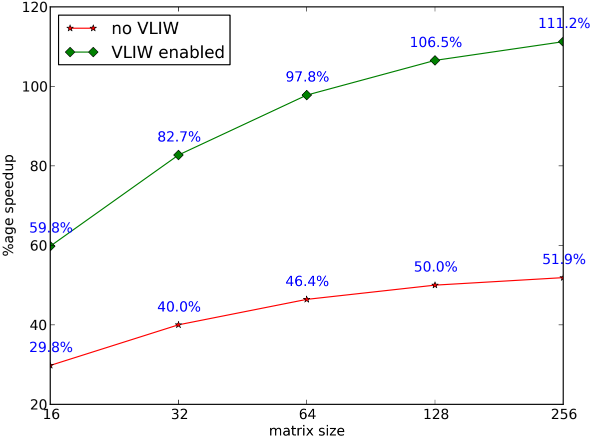
- propagation of data constants into the binary code
- loop unrolling

```
// generate kernel
mac = gen_mac(&A, &B, &C);

clear(C)
for (y=0; y < n; y++){
  for (i=0; i < p; i+=VSIZE){
    mac(y,i);// execute kernel
} }
```

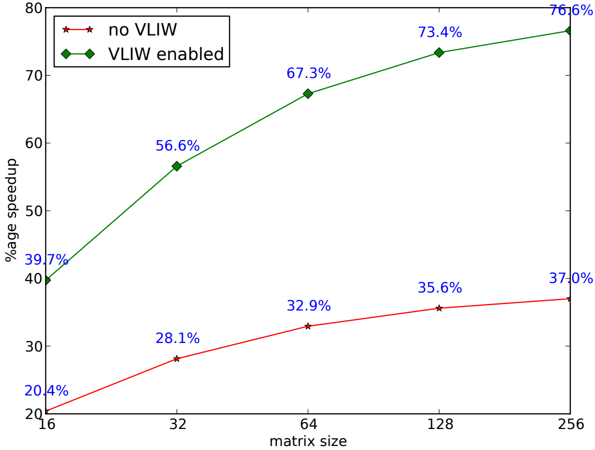


experiment #1: results (integer data)



- STxP70-v4 core, simulation with `sxrun` **in CAS mode**
- `-O3`, `-funroll-loops -Munroll -Mconfig=mult:yes`,
- VLIW enabled with `-Mconfig=vliw:dualcoreALU`

experiment #1: results (float data)



- STxP70-v4 core, simulation with `sxrun` **in ISS mode**
- `-O3`, `-funroll-loops -Munroll -Mconfig=mult:yes`,
- VLIW enabled with `-Mconfig=vliw:dualcoreALU`

experiment #2 with `deGoal`

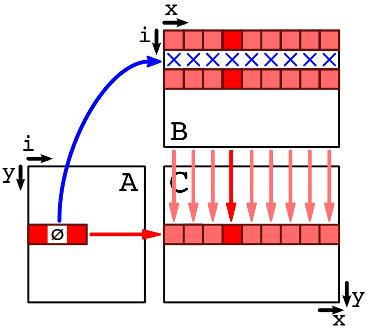
Objective:

- same as #1
- code generation depending on the **data values in** `A`

```
clear(C)

// generate the generic section
mac=gen_mac(prog_buf,&A,&B,&C);

// process matrix multiplication
for (y=0; y < n; y++){
  for (i=0; i < p; i+=VSIZE){
    // do data specialization
    mac=gensp_mac(prog_buf,y,i);
    if (NULL != mac)
      mac(y, i);
} }
```



experiment #2 with deGoal

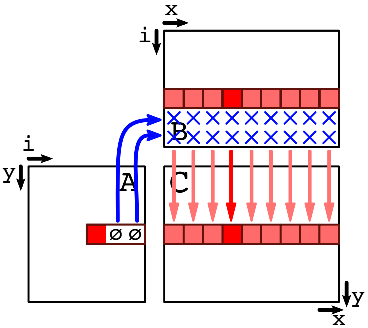
Objective:

- same as #1
- code generation depending on the **data values in** A

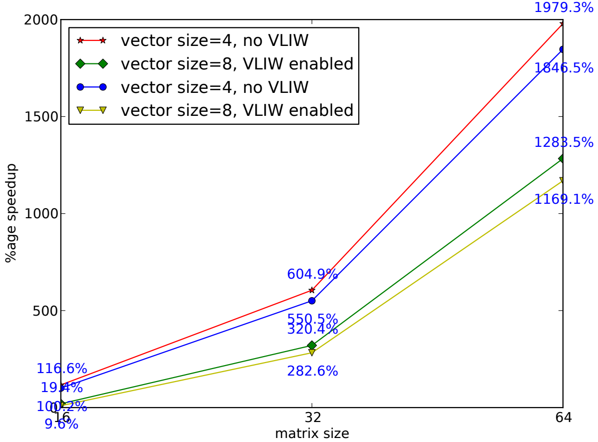
```
clear(C)

// generate the generic section
mac=gen_mac(prog_buf,&A,&B,&C);

// process matrix multiplication
for (y=0; y < n; y++){
  for (i=0; i < p; i+=VSIZE){
    // do data specialization
    mac=gensp_mac(prog_buf,y,i);
    if (NULL != mac)
      mac(y, i);
} }
```



experiment #2: results (integer data)



- STxP70-v4 core, simulation with `sxrun` **in CAS mode**
- same compilation options as experiment #1
- results displayed = best case: matrix A is the identity matrix

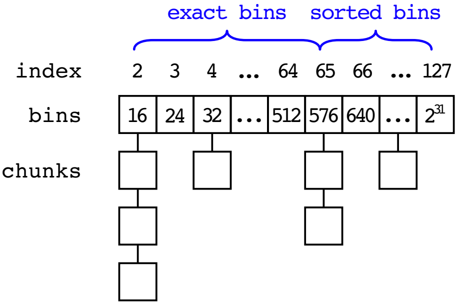
Outline

an overview of `dlmalloc`

reference: `dlmalloc`, main implementation in GNU/Linux systems

- `chunk`: reference to a fragment of free memory
- `bins`: array of linked lists
- the `bin_hash` function associates a fragment size to a index (binary tree)

data structure used in `dlmalloc` (stolen from[a]):



[a] D. Lea: A memory allocator, http://g.oswego.edu/dl/html/malloc.html, 2000.

malloc processing:

1. `index = bin_hash(size)`
2. `return best_fit(bins[index], size)`

requirements

. . . for a memory allocator targeting many-core embedded systems:

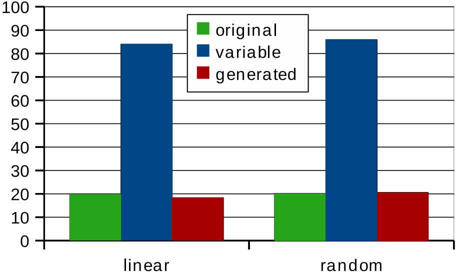
- ability to deal with multiple physical memories (TCMs, intra-cluster shared mem, inter-cluster memory . . .)
- a multi-instanciable allocator, each instance being parameterized according to its execution context
- a configurable `bin_hash` function; the hash table depends on:
  - the architecture / memory organization
  - the application profile

for further details, follow Yves's presentation

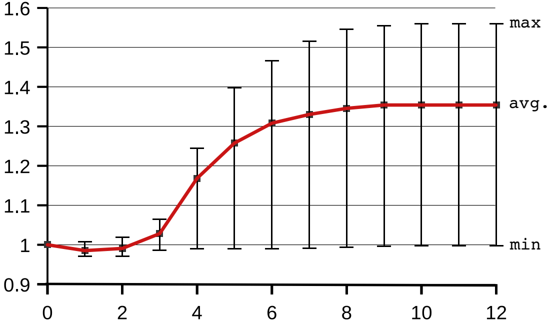
"A Portable Runtime for different P2012 hardware flavors"
16:15 – room 225

performance results

Execution time of the hash function (nb. cycles):



Speedup factor:
(min/average/max)
over number of successive code generations



Conclusion

Current results for P2012:

- ported the STxP70 v3 & v4 cores
- FPx and VLIW support (STxP70-4)

Other architectures currently supported by `deGoal`:

- ARM thumb (T1-4), 32 bits (ARM32_1-3), NEON SIMD
- MIPS (host processor of CEA-LETI GENEPY architecture)
- CUDA PTX assembly language
- Other NDA-based ABI architectures (RISC/VLIW)

leti

LABORATOIRE D'ÉLECTRONIQUE
ET DE TECHNOLOGIES
DE L'INFORMATION

list

LABORATOIRE D'INTÉGRATION
DES SYSTÈMES
ET DES TECHNOLOGIES

Thanks!
Questions?