REPARA: Reeengineering for Heterogeneous Parallelism for Performance and Energy in C++

J. Daniel Garcia

Computer Architecture Group.
Universidad Carlos III de Madrid

January 21, 2015

1 Introduction

2 Source code preparation

3 Application partitioning

4 Transformation analysis

5 From attributes to run-time

6 Summary

Context

- The end for the **free-lunch** era.

Context

- The end for the **free-lunch** era.
- New architectures with diversity in computing elements.
  - Multi-cores, GPUs, DSPs, FPGAs.

Context

- The end for the **free-lunch** era.
- New architectures with diversity in computing elements.
  - Multi-cores, GPUs, DSPs, FPGAs.
- A switch of focus.
  - From performance centric serial computations, ...
  - ...to energy efficient parallel computations.

Context

- The end for the **free-lunch** era.
- New architectures with diversity in computing elements.
  - Multi-cores, GPUs, DSPs, FPGAs.
- A switch of focus.
  - From performance centric serial computations, ...
  - ...to energy efficient parallel computations.
- Programming heterogeneous parallel architectures:
  - Lack of unified programming model for diverse devices.
  - Need to maximize performance and energy efficiency.
  - Costly development process porting to multiple devices.
  - Need to modernize existing legacy code bases.

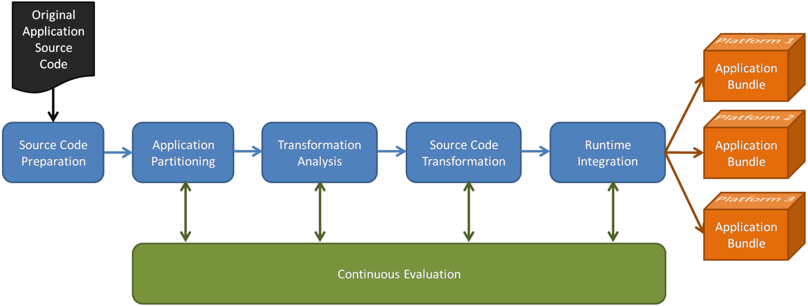
The REPARA Vision

Vision

The **REPARA** project aims to help in the transformation and deployment of new and legacy applications in **parallel heterogeneous computing architectures** while maintaining balance between:

- *application performance*,
- *energy efficiency*, and
- *source code maintainability*.

A workflow for code transformations



Problems to be solved (I)

- **Source code preparation**:
  - Adaptation of legacy code.
  - Enforcement of C++ subset (REPARA-C++).

Problems to be solved (I)

- **Source code preparation**:
  - Adaptation of legacy code.
  - Enforcement of C++ subset (REPARA-C++).
- **Application partitioning**:
  - Describe the target platform.
  - Map software components to specific computing devices.

Problems to be solved (I)

- **Source code preparation**:
  - Adaptation of legacy code.
  - Enforcement of C++ subset (REPARA-C++).
- **Application partitioning**:
  - Describe the target platform.
  - Map software components to specific computing devices.
- **Transformation analysis**:
  - Identify transformation opportunities.
  - Generation of an Abstract Intermediate Representation (REPARA-AIR).

Problems to be solved (II)

- **Source code transformation**:
  - Interactive and non-interactive refactoring.
  - Automated transformation to FPGA.

Problems to be solved (II)

- **Source code transformation**:
  - Interactive and non-interactive refactoring.
  - Automated transformation to FPGA.
- **Runtime engineering**:
  - Coordination of software components (FastFlow).
  - Manage statically and dynamically partitioned applications.

Problems to be solved (II)

- **Source code transformation**:
  - Interactive and non-interactive refactoring.
  - Automated transformation to FPGA.
- **Runtime engineering**:
  - Coordination of software components (FastFlow).
  - Manage statically and dynamically partitioned applications.
- **Continuous evaluation**:
  - Prediction and monitoring of performance and energy.
  - Evalutation of software maintainability.

1 Introduction

2 Source code preparation

3 Application partitioning

4 Transformation analysis

5 From attributes to run-time

6 Summary

Legacy code

- C++ is an evolving language, but highly backwards compatible.
  - And it has a C subset.

Legacy code

- C++ is an evolving language, but highly backwards compatible.
  - And it has a C subset.

- In the context of parallel heterogeneous architectures a software component may run at:
  - **Host side**: Runs at the CPU.
    - Supports all the ISO C++11 features.
  - **Device side**: Runs on device (GPU, FPGA, DSP).
    - Restricted subset from ISO C++11.

GPU: Examples

- Forbid the use of *bit-fields* data members in structures.
- Disallow the use of VLAs and *flexible* array data members.
- GPU software components cannot use dynamic type binding.
  - `virtual` member functions.
  - Virtual inheritance.
- Memory management through `new`/`delete` cannot be used.
- GPU software components cannot make use of exceptions.
- Restricted version of the C++ standard library.

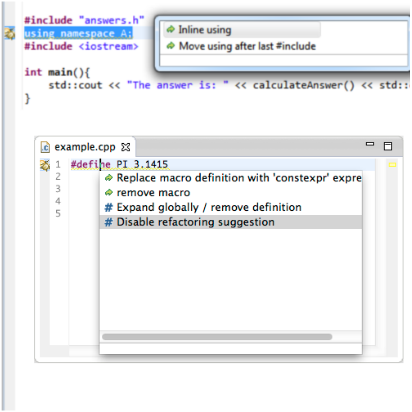
FPGA: Examples

- System calls are not supported.
- Pointer casting is not allowed unless it is between native C types.
- Recursive functions are not allowed.
- Arrays of pointers are supported only if each pointer points to a scalar or array of scalars.
- Dynamic memory management is not supported.

Cevelop: An IDE for REPARA

http://www.cevelop.com

- Based on Eclipse IDE.
- Already includes some refactorings for C/C++ improvement.
- Future versions:
  - Application partitioning.
  - Transformation analysis.
  - Source code transformation.



1 Introduction

2 Source code preparation

3 Application partitioning

4 Transformation analysis

5 From attributes to run-time

6 Summary

Platform descriptions

- **HPP-DL**: A description language to represent all elements from a parallel heterogeneous platform.
  - JSON based.

- A platform description includes information about hardware and other platform specific information (e.g. I/O ports, IRQs, . . .).

HPP-DL: example

```
{
/* Metainformation of HPP −DL */
" class ": "hpp",
" description ": " Human readable description ",
" version ": "1.0",
" date ": " 2014 −01 −13 10:00",
" components ":[
/* Definition of harware platform */
{
" class ": " platform ",
"id": " platform :0",
" description ": " REPARA Reference System . X9DRG −QF (To be filled by O.E.M.)",
" model ": "X9DRG −QF",
" vendor ": " Supermicro Inc.",
" numa_nodes ": 2,
" processors ": 2,
" cores ": 4,
" pu_num ": 8,
" global_mem_size ": "16 GiB",
" capabilities ": []
},
/* Definition of processor 0 */
{
" class ": " processor ",
...
```

Kernel anotations through attributes

- Source code can be annotated to identify kernels from an application.
  - Many of this annotations can be automatically generated.
  - May be manually refined.

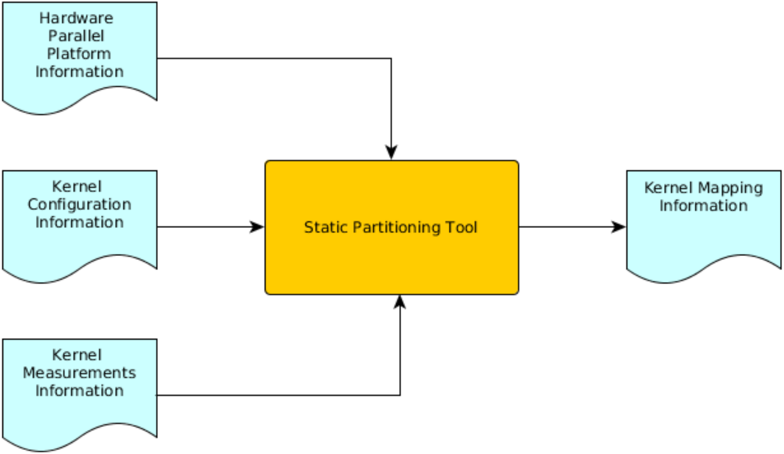
Kernel anotations through attributes

- Source code can be annotated to identify kernels from an application.
  - Many of this annotations can be automatically generated.
  - May be manually refined.
- C++ offers attributes as an alternate to traditional *pragmas*.
  - Less verbose annotation mechanism in some situations.
  - Better integration with language syntax.

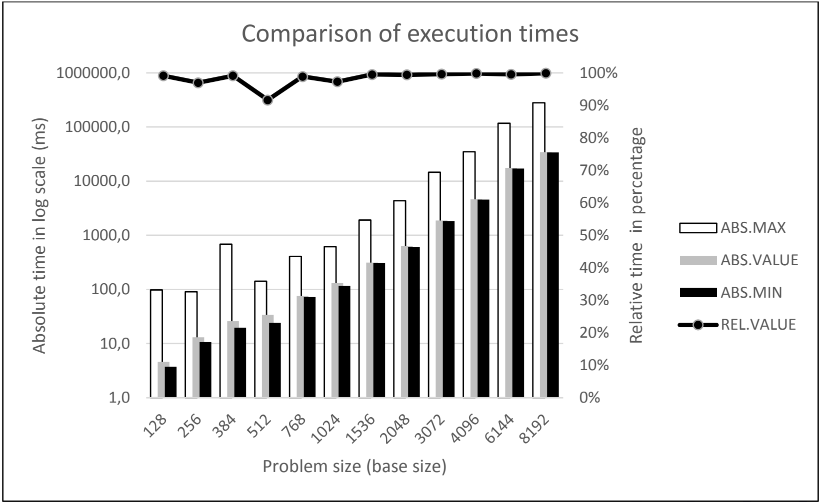
Example

```
[[ rpr::kernel, rpr::target(CPU,GPU), rpr::in(A,B,n,data), rpr::out(C)]]
for (int i=0; i<n; ++i)
  for (int j=0; j<i; ++j)
    C[i] = A[i] * B[j] + data;
```

Static partitioning



Evaluation: Transitive closure



1 Introduction

2 Source code preparation

3 Application partitioning

4 Transformation analysis

5 From attributes to run-time

6 Summary

Identifying opportunities

- Kernels can be identified:
  - Manually by the programmer.
  - By an automated tool.

Identifying opportunities

- Kernels can be identified:
  - Manually by the programmer.
  - By an automated tool.
- Identifying a kernel requires:
  - Identify input and output parameters.
  - Identify target devices where the kernel is valid.
  - Identify size parameters.
- Additionally:
  - More sophisticated properties.
  - Patterns (e.g. pipeline, farm, . . .).

An Abstract Intermediate Representation

- Transformations of software components to multiple programming models.
  - Adopt common strategy in compiler technology: a front-end and a back-end.

An Abstract Intermediate Representation

- Transformations of software components to multiple programming models.
  - Adopt common strategy in compiler technology: a front-end and a back-end.
- **front-end**:
  - Identifies transformation opportunities.
  - Adds meta-data to original source code.
  - Generates an abstract intermediate representation.

An Abstract Intermediate Representation

- Transformations of software components to multiple programming models.
  - Adopt common strategy in compiler technology: a front-end and a back-end.
- **front-end**:
  - Identifies transformation opportunities.
  - Adds meta-data to original source code.
  - Generates an abstract intermediate representation.
- **back-end**:
  - Multiple back-ends to transform to different programming models.
  - One additional back-end for FPGA.

1 Introduction

2 Source code preparation

3 Application partitioning

4 Transformation analysis

5 From attributes to run-time

6 Summary

Simple kernels

```
int main() {
  constexpr size_t size_a = 256, size_b = 32;
  std::vector<long> A(size_a), B(size_b);

  [[ rpr::kernel, rpr::in(A[]), rpr::out(A[]), rpr::target(CPU)]]
  for (size_t i=0; i<A.size(); ++i)
    A[i] = F(i);

  [[ rpr::kernel, rpr::in(B[]), rpr::out(B[]), rpr::target(CPU)]]
  for (size_t i=0; i<B.size(); ++i)
    B[i] = G(i);

  [[ rpr::kernel, rpr::in(A[], B[]), rpr::out(x), rpr::target(CPU)]]
  long x = H(A,B);

  std::cout << x << std::endl;
  return 0;
}
```

Transformation of simple kernels

```
int main() {
  constexpr size_t size_a = 256, size_b = 32;
  std::vector<long> A(size_a), B(size_b);

  ff::ParallelFor pf;
  pf.parallel_for(0, A.size(), [&A](long i) {
    A[i] = F(i);
  });

  pf.parallel_for(0, B.size(), [&B](long i) {
    B[i] = G(i);
  });

  long x = H(A,B);
  std::cout << x << std::endl;
  return 0;
}
```

Matrix Vector multiplication

```
int main() {
  vector<float> M(16*1024), V(1024), R(16);

  [[ rpr::kernel, rpr::out(M[16*1024]), rpr::target(CPU)]]
  for (size_t i=0; i<16; ++i)
    for (size_t j=0; j<1024; ++j)
      M[i*1024+j] = static_cast<float>(i*1024+j+1);

  [[ rpr::kernel, rpr::out(V[1024]), rpr::target(CPU)]]
  for (size_t i=0; i<1024; ++i) V[j]=1.0;

  [[ rpr::kernel, rpr::in(M[16*1024], V[1024]), rpr::out(R[16]), rpr::target(GPU)]]
  for (size_t i=0; i<16; ++i) {
    float sum = 0.0;
    for (size_t j=0; j<1024; ++j)
      sum += M[i*1024+j] * V[j];
    R[i] = sum;
  }

  print_result(R,16);
}
```

Pipelines

```
...
[[ rpr :: pipeline,  rpr ::stream(B,C)]]
for  ( size_t  y=0; y<MAX; ++y) {

  [[ rpr :: kernel,  rpr :: out(B[]) ]]
  for  ( size_t  i=0;  i<N; ++i)
    for  ( size_t  j=0;  j<N; ++j)
      B[i*N + j]  = y  + float{i+j};

  [[ rpr :: kernel,  rpr :: in (B[]) ,  rpr :: out(C[]) ]]
  for  ( size_t  i=0;  i<N; ++i)
    for  ( size_t  j=0;  j<N; ++j)
      for  ( size_t  k=0;  j<N; ++k)
        C[i*N+j] = A[i*N+k] * B[k*N+j]

  [[ rpr :: kernel,  rpr :: in (C[]) ,  rpr :: out(R[]) ]]
  for  ( size_t  i=0;  i<N; ++i)
    for  ( size_t  j=0;  j<N; ++j)
       R[i*N+j] = C[j*N+i]
}
```

Farm

```
[[ rpr :: pipeline , rpr :: stream(B,C)]]
for ( size_t y=0; y<MAX; ++y) {

  [[ rpr :: kernel , rpr :: out(B []) , rpr :: farm(2) ]]
  for ( size_t i=0; i<N; ++i)
    for ( size_t j=0; j<N; ++j)
      B[i*N + j] = y + float{i+j};

  [[ rpr :: kernel , rpr :: in (A [], B []) , rpr :: out(C []) , rpr :: farm() ]]
  for ( size_t i=0; i<N; ++i)
    for ( size_t j=0; j<N; ++j)
      for ( size_t k=0; j<N; ++k)
        C[i*N+j] = A[i*N+k] * B[k*N+j]

  [[ rpr :: kernel , rpr :: in (C []) , rpr :: out(R []) ]]
  for ( size_t i=0; i<N; ++i)
    for ( size_t j=0; j<N; ++j)
      R[i*N+j] = C[j*N+i]
}
```

Map

```
[[ rpr :: pipeline, rpr :: stream(B,C)]]
for ( size_t y=0; y<MAX; ++y) {

  [[ rpr :: kernel, rpr :: out(B[]) , rpr :: map(2)]]
  for ( size_t i=0; i<N; ++i)
    for ( size_t j=0; j<N; ++j)
      B[i*N + j] = y + float{i+j};

  [[ rpr :: kernel, rpr :: in (A[],B[]) , rpr :: out(C[]) , rpr :: map()]]
  for ( size_t i=0; i<N; ++i)
    for ( size_t j=0; j<N; ++j)
      for ( size_t k=0; j<N; ++k)
        C[i*N+j] = A[i*N+k] * B[k*N+j]

  [[ rpr :: kernel, rpr :: in (C[]) , rpr :: out(R[]) ]]
  for ( size_t i=0; i<N; ++i)
    for ( size_t j=0; j<N; ++j)
      R[i*N+j] = C[j*N+i]
}
```

Reductions

```
long r = std::numeric_limits<long>::min();

[[ rpr::kernel, rpr::in(A[]), rpr::reduce(max,r)]]
for (size_t i=0; i<N; ++i)
  r = std::max(A[i], r);

std::cout << r << std::endl;
```

Asynchronous execution

```
[[ rpr::kernel, rpr::async]]
for (size_t i=0;i<N; ++i)
  for (size_t j=0; j<N; ++j) {
    A[i*N+j] = float{i+j};
    B[i*N+j] = float{abs(j-i)};
  }

[[ rpr::kernel, rpr::async]]
for (size_t i=0; i<N; ++i) V[i] = float{i};

[[ rpr::sync]] ; // Explicit sync
```

Asynchronous execution

```
[[ rpr :: kernel, rpr :: async]]
for ( size_t i=0;i<N; ++i)
  for ( size_t j=0; j<N; ++j) {
    A[i*N+j] = float{i+j};
    B[i*N+j] = float{abs(j-i)};
  }

[[ rpr :: kernel, rpr :: async]]
for ( size_t i=0; i<N; ++i) V[i] = float{i};

[[ rpr :: kernel]] // Implicit sync
f () ;
```

1 Introduction

2 Source code preparation

3 Application partitioning

4 Transformation analysis

5 From attributes to run-time

6 Summary

Summary

- Performance, energy efficiency and source code maitainability need to be balanced.
- Legacy code needs to be considered.
  - Much more legacy code than new code out there.
- Refactoring C++ code to:
  - Enforce specific device rules.
  - Apply transformations to specific programming models.
  - Generate FPGA code.
- Application partitioning using HW description, kernel measurements and code.
- Transformation combining front-end and back-end.
- C++ attributes to enrich code with annotations.

REPARA: Reeengineering for Heterogeneous Parallelism for Performance and Energy in C++

J. Daniel Garcia

Computer Architecture Group.
Universidad Carlos III de Madrid

January 21, 2015