LAPP
Laboratoire d'Annecy de Physique des Particules

Introduction : Compilation

Pierre Aubert

ESCAPE
cta cherenkov telescope array
cnrs
UNIVERSITÉ SAVOIE MONT BLANC

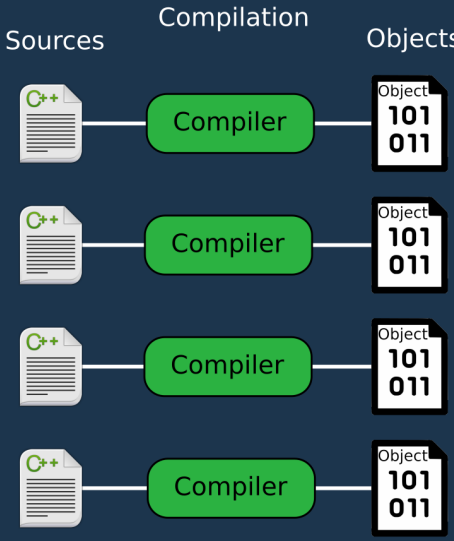
Compilation : From Text to Binary

Compilation : From Text to Binary

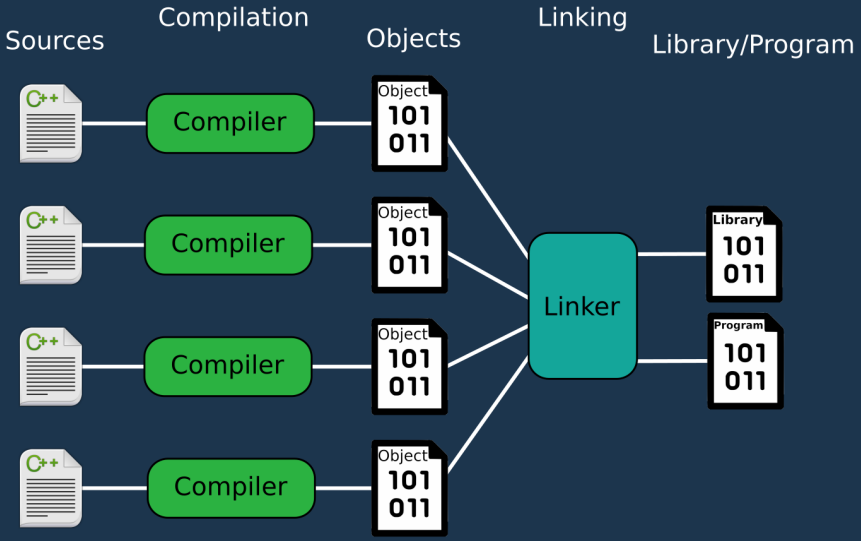
Sources



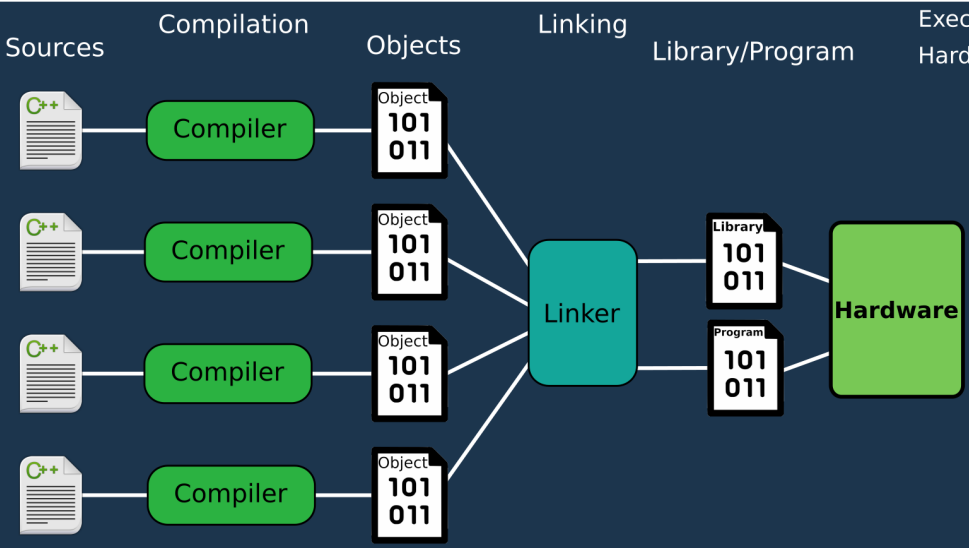

Compilation : From Text to Binary



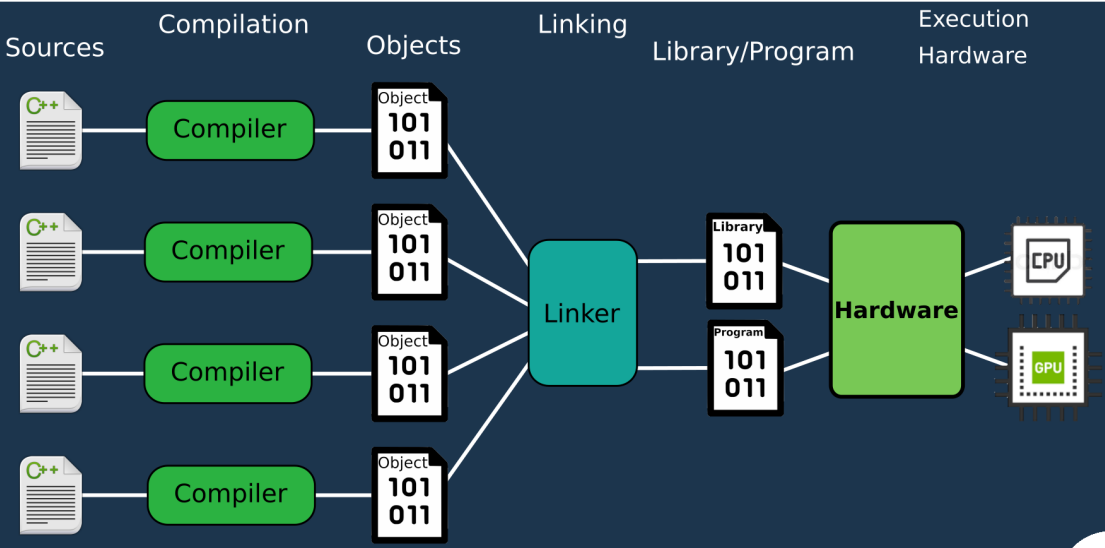
Compilation : From Text to Binary



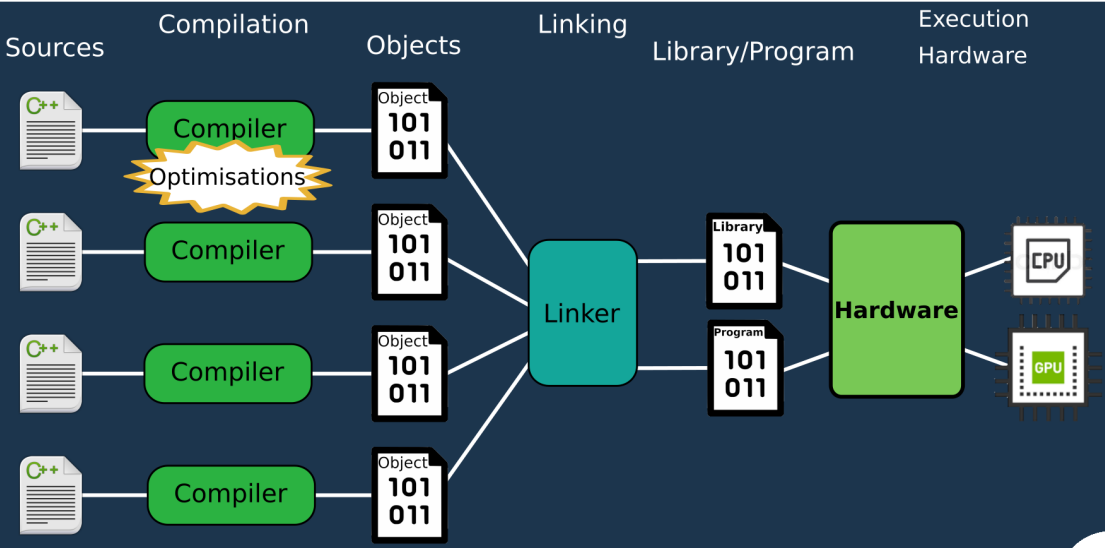
Compilation : From Text to Binary



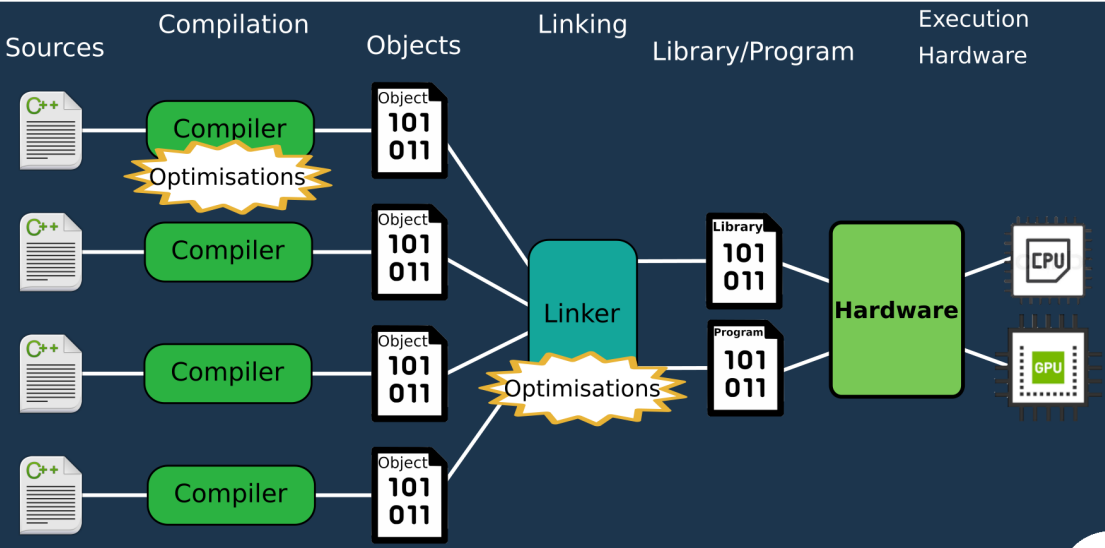
Compilation : From Text to Binary



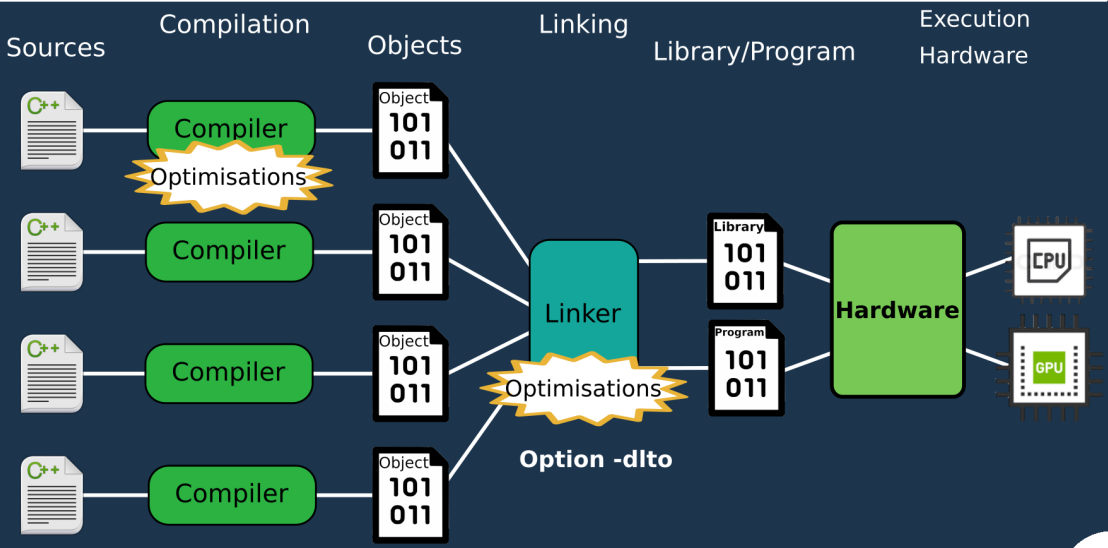
Compilation : From Text to Binary



Compilation : From Text to Binary



Compilation : From Text to Binary



Compiler workflow

Compiler workflow

Languages

Fortran

Compiler workflow

Languages

Fortran

Binaries

Compiler workflow

Languages

Front-End

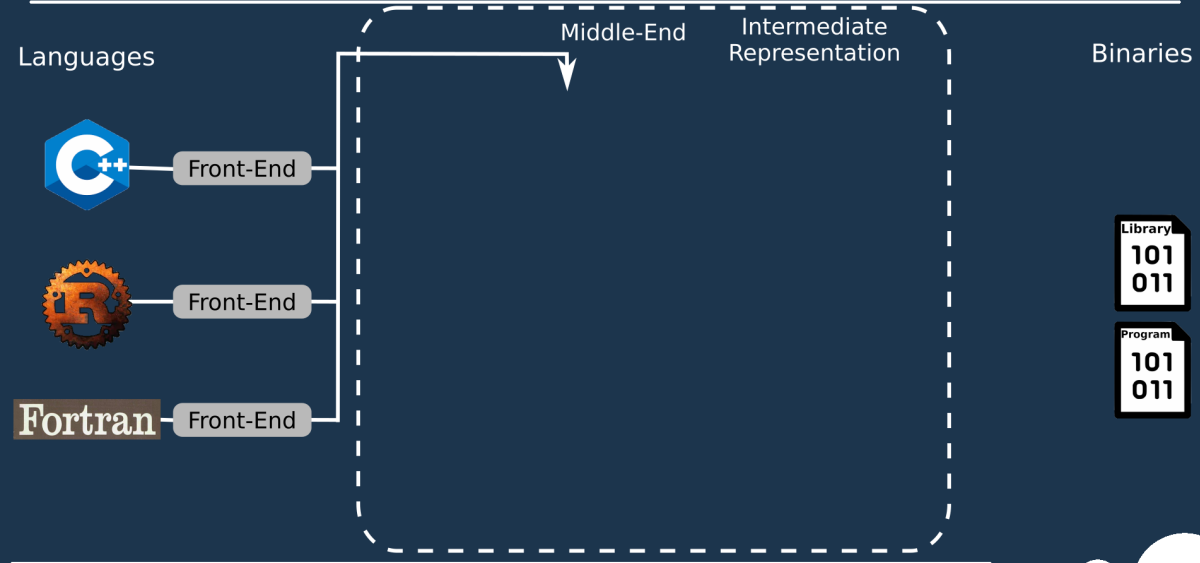
Front-End

Fortran

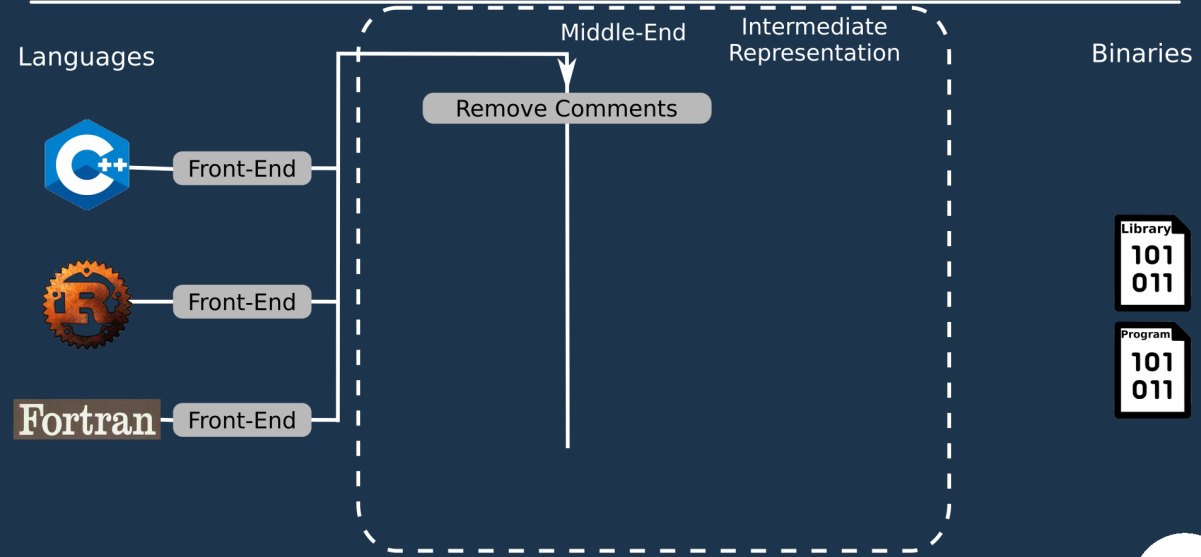
Front-End

Binaries

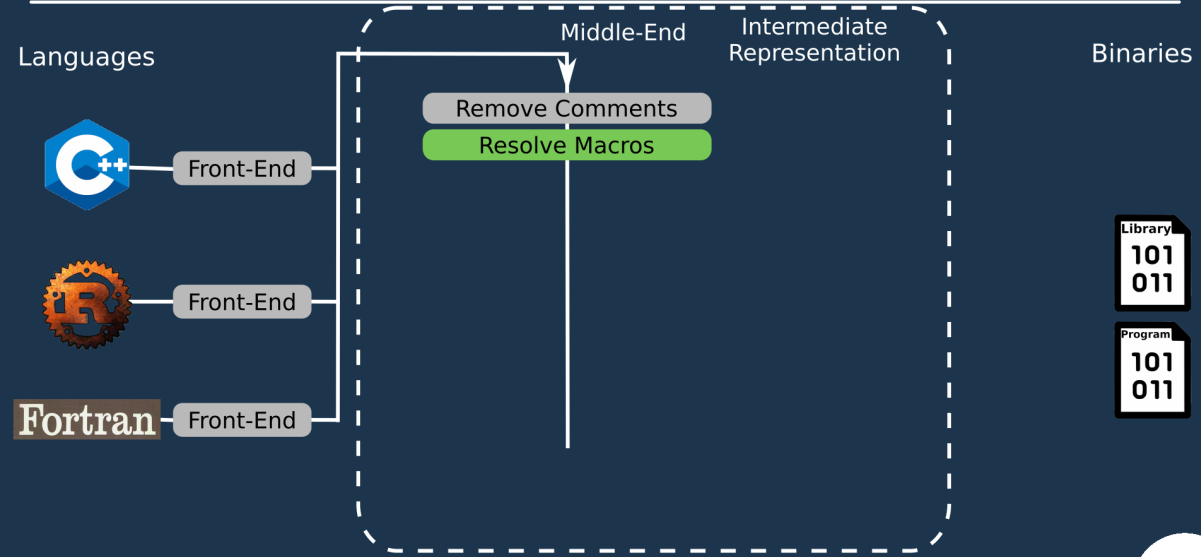
Compiler workflow



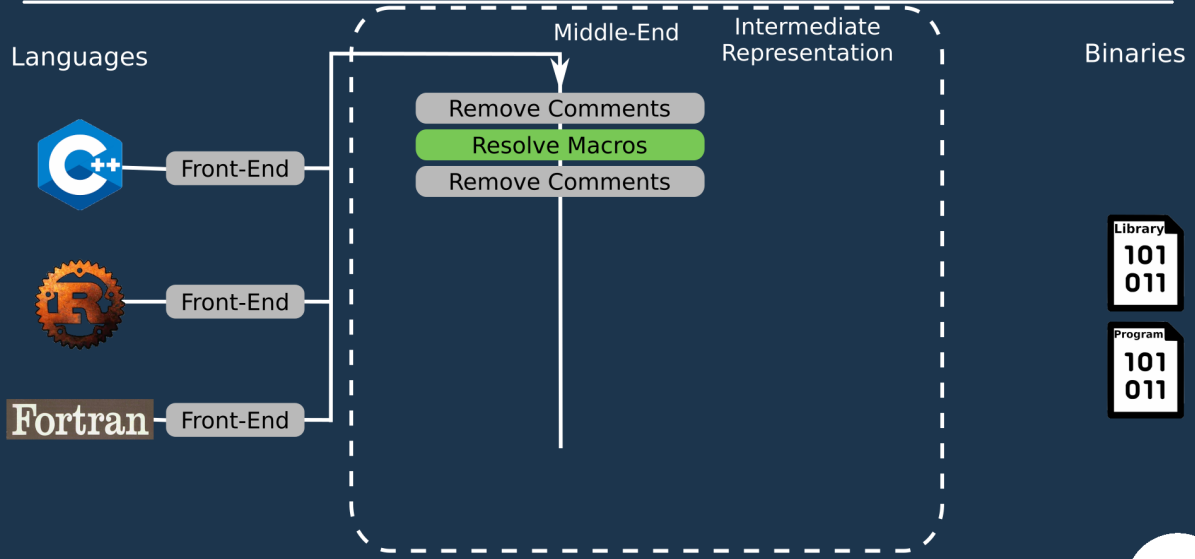
Compiler workflow



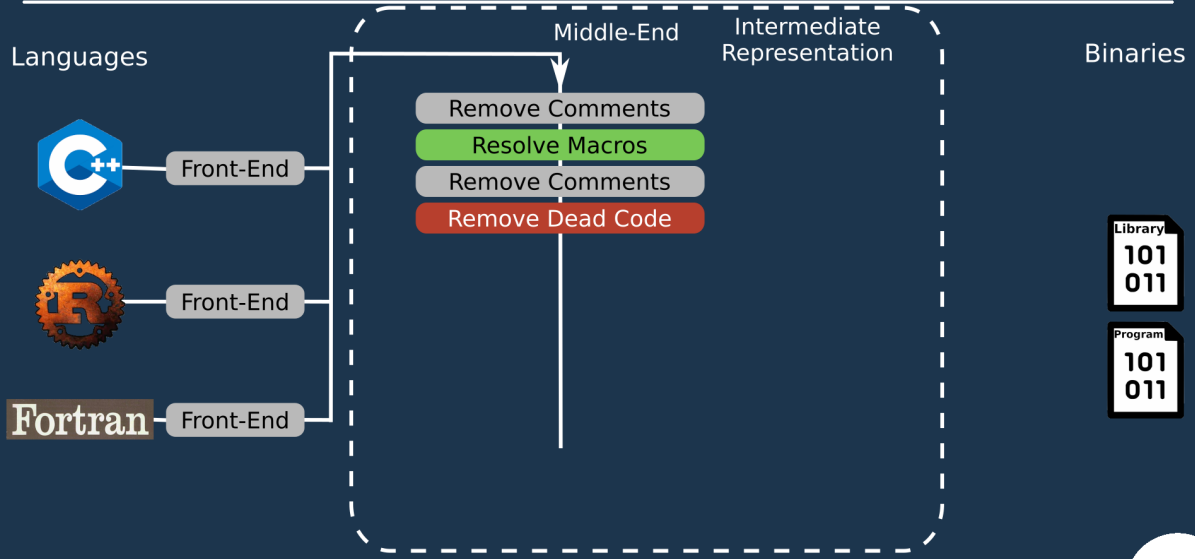
Compiler workflow



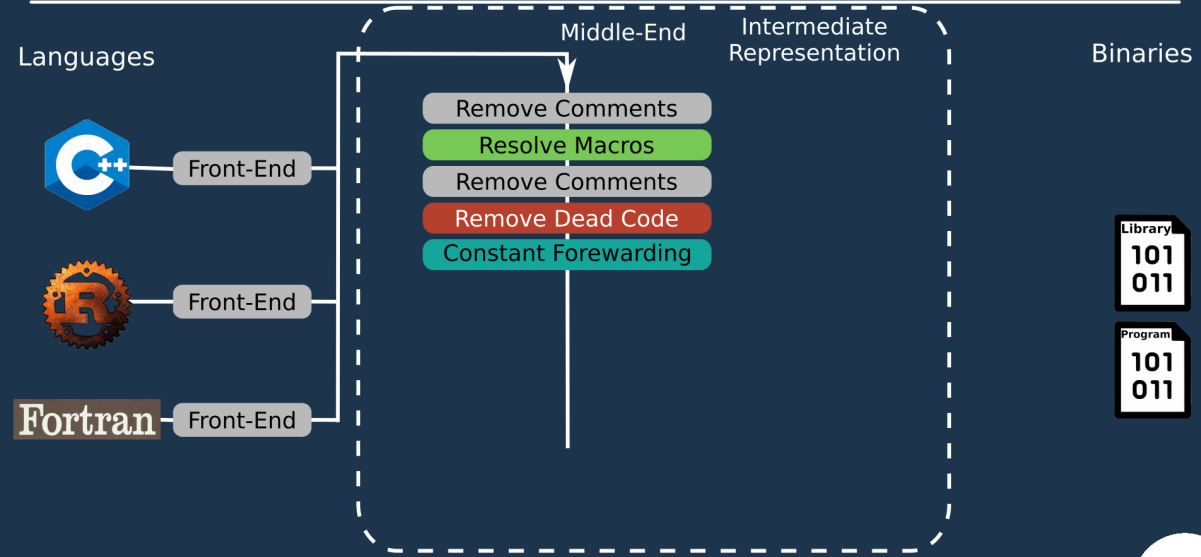
Compiler workflow



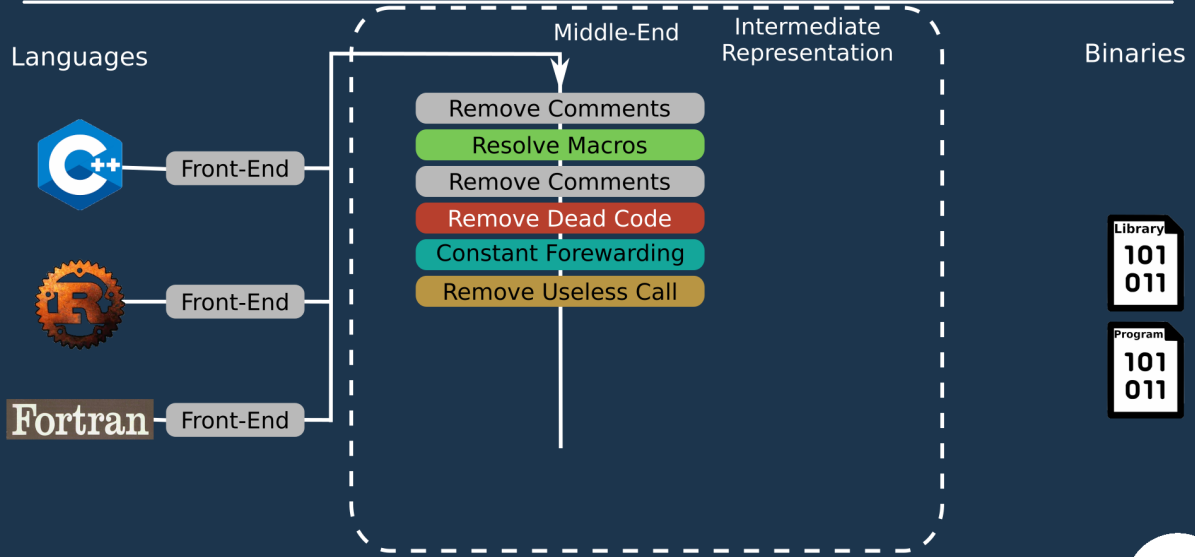
Compiler workflow



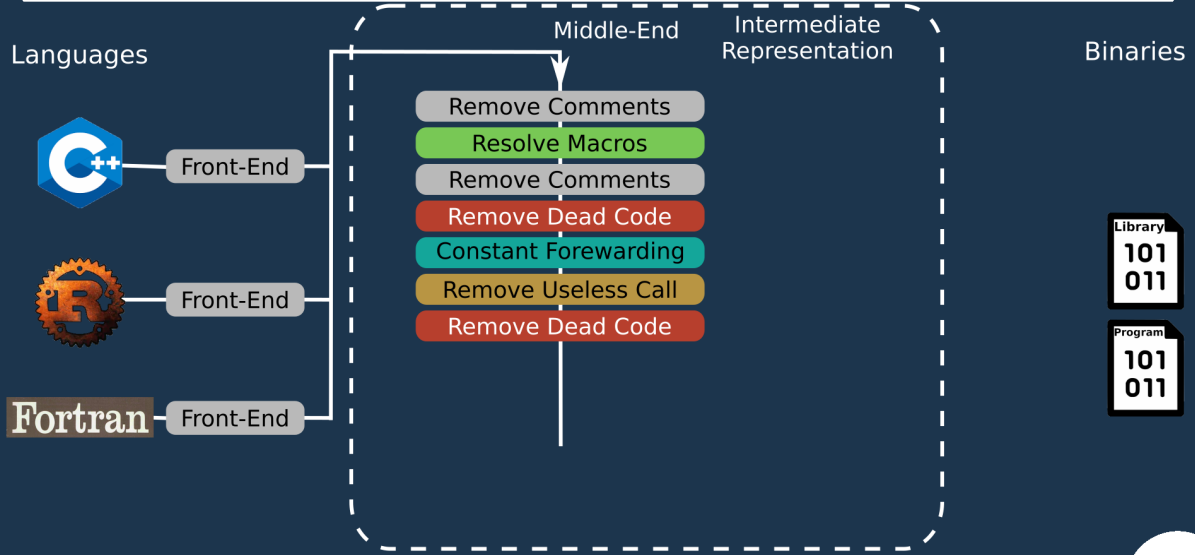
Compiler workflow



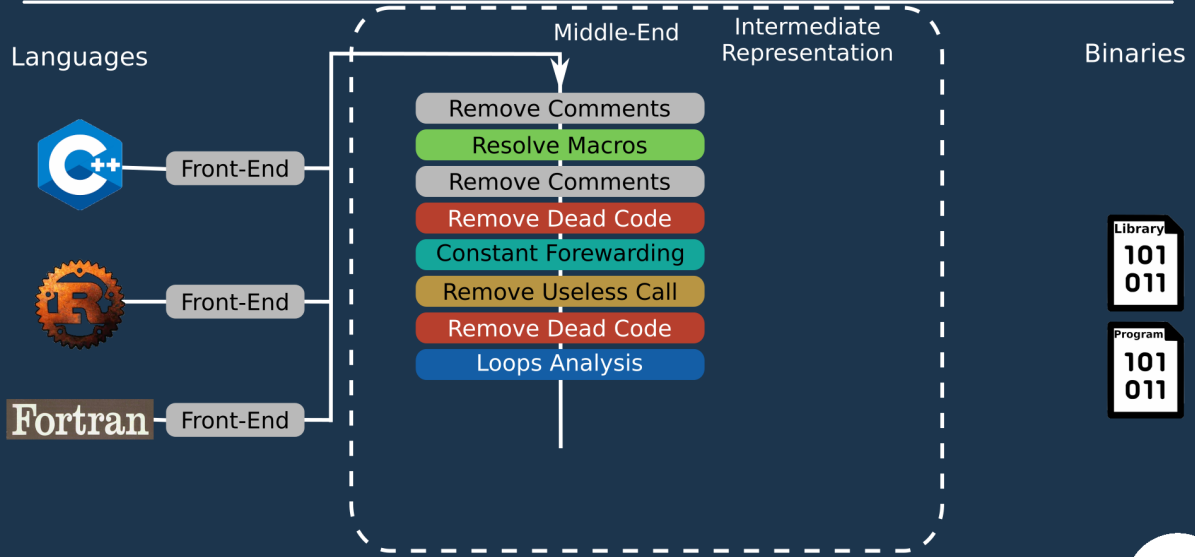
Compiler workflow



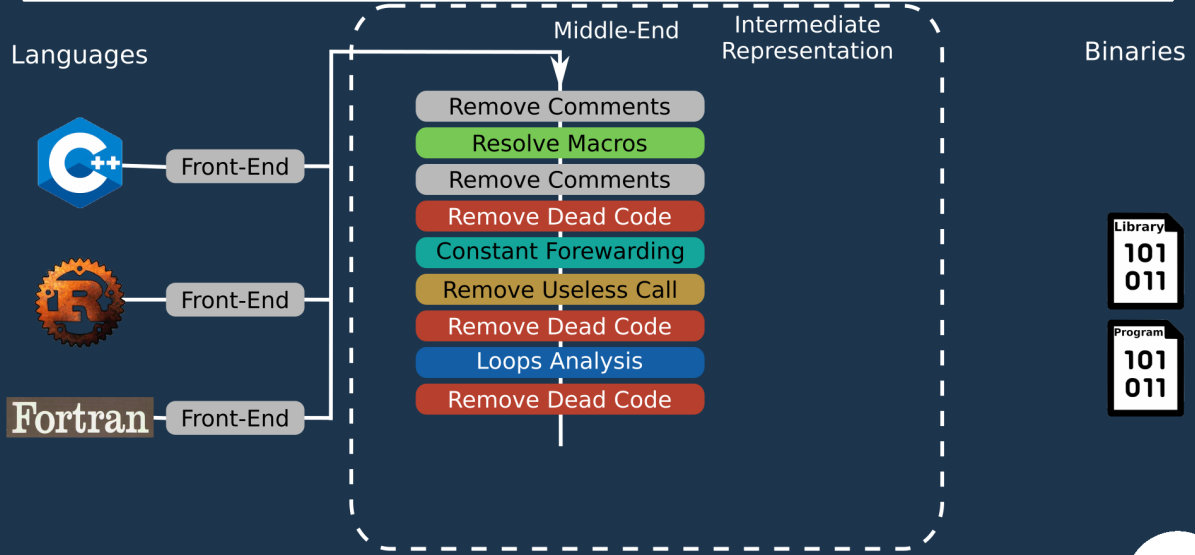
Compiler workflow



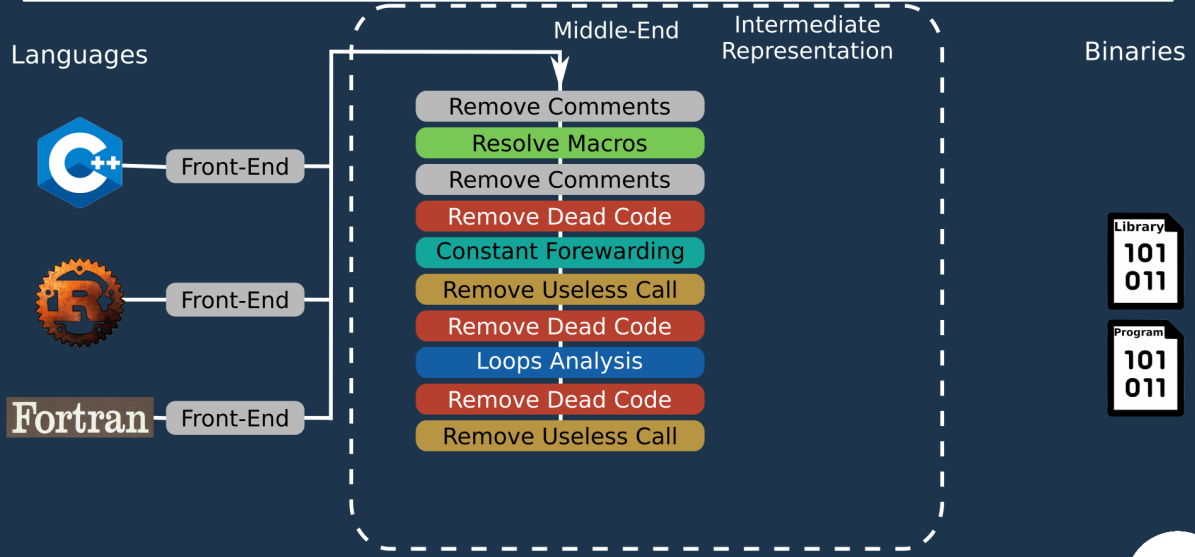
Compiler workflow



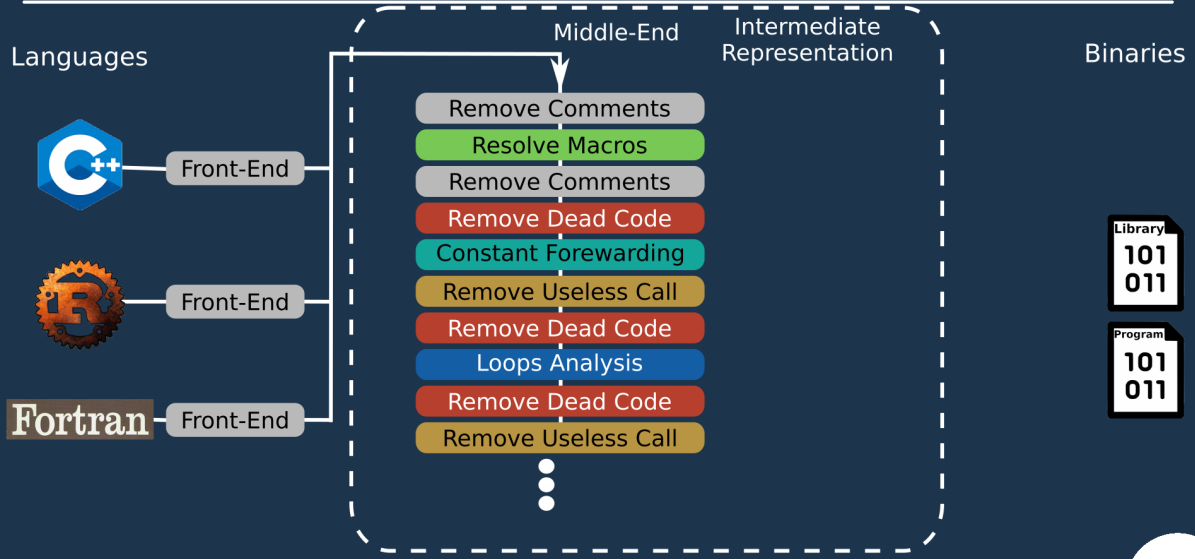
Compiler workflow



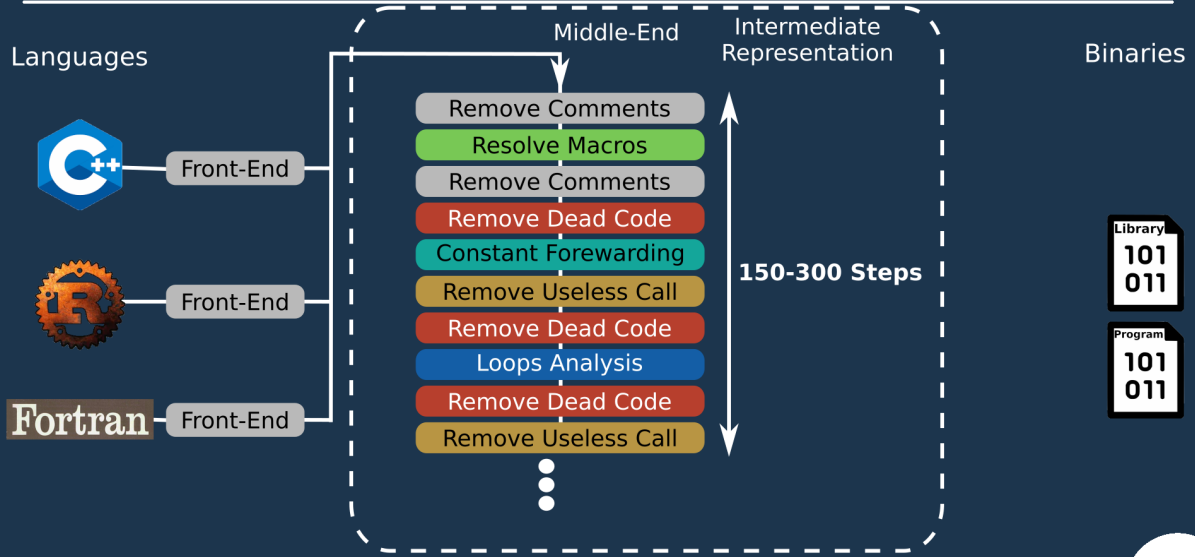
Compiler workflow



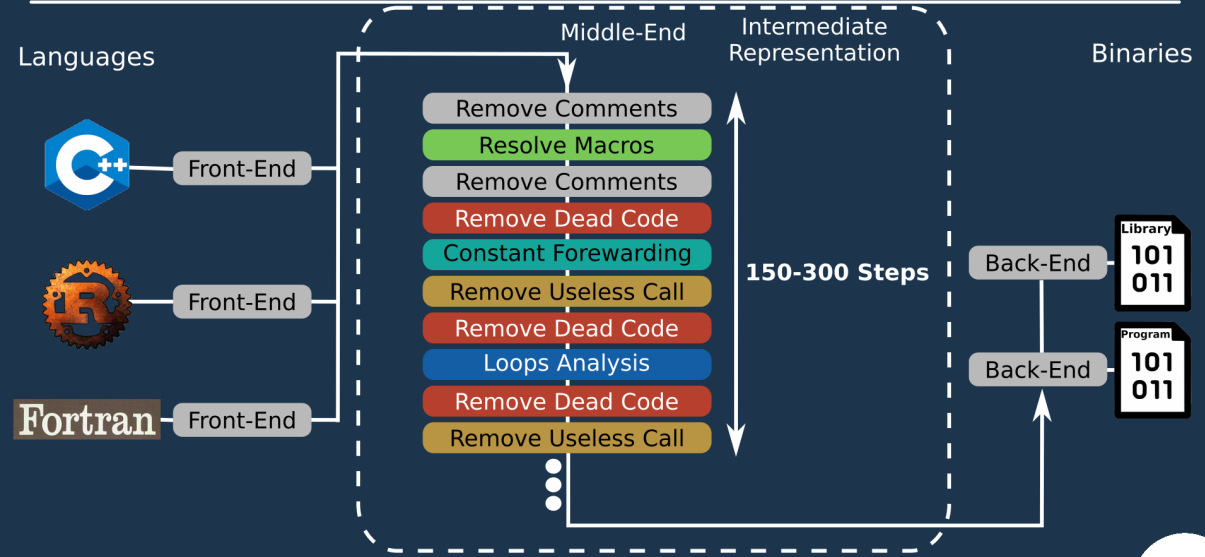
Compiler workflow



Compiler workflow



Compiler workflow



Compiler intermediate representation

Compiler intermediate representation

Program

```
float square(float x){
»       return x*x;
}
int main(){
»       float y(42.0f);
»       std::cout << square(y) << std::endl;
»       return 0;
}
```

Compiler intermediate representation

Program

```
float square(float x){
»       return x*x;
}
int main(){
»       float y(42.0f);
»       std::cout << square(y) << std::endl;
»       return 0;
}
```

Graph

Compiler intermediate representation

Program

```
float square(float x){
»       return x*x;
}
int main(){
»       float y(42.0f);
»       std::cout << square(y) << std::endl;
»       return 0;
}
```

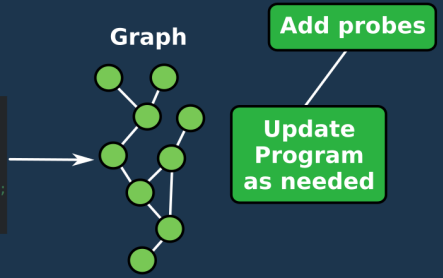
Graph

Update Program as needed

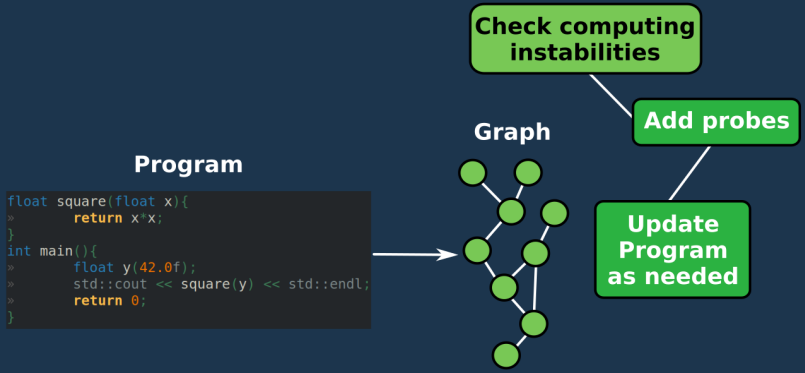
Compiler intermediate representation

Program

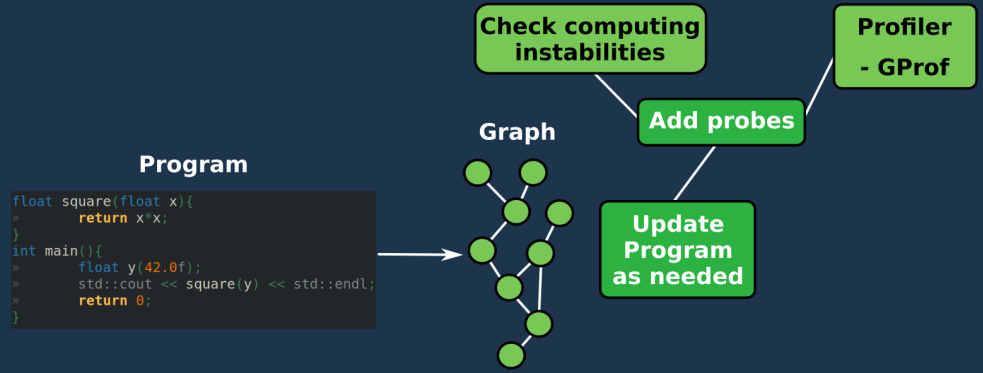
```
float square(float x){
»       return x*x;
}
int main(){
»       float y(42.0f);
»       std::cout << square(y) << std::endl;
»       return 0;
}
```



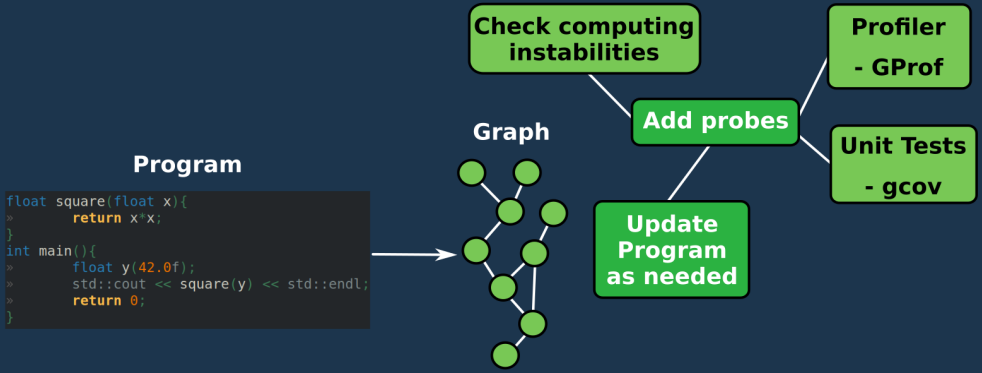
Compiler intermediate representation



Compiler intermediate representation



Compiler intermediate representation



Compiler intermediate representation

Program

```
float square(float x){
»       return x*x;
}
int main(){
»       float y(42.0f);
»       std::cout << square(y) << std::endl;
»       return 0;
}
```

Check computing instabilities

Profiler - GProf

Add probes

Unit Tests - gcov

Update Program as needed

Counters on lines and branches

Compiler intermediate representation

Program

```
float square(float x){
»       return x*x;
}
int main(){
»       float y(42.0f);
»       std::cout << square(y) << std::endl;
»       return 0;
}
```

Check computing instabilities

Profiler
- GProf

Add probes

Unit Tests
- gcov

Counters on lines and branches

Update Program as needed

Optimisation
- Constant forewarding
- Inlining
- Auto-vectorization
- Auto-parallelization

Compiler intermediate representation

Program

```
float square(float x){
»       return x*x;
}
int main(){
»       float y(42.0f);
»       std::cout << square(y) << std::endl;
»       return 0;
}
```

Graph

Check computing instabilities

Profiler
- GProf

Add probes

Unit Tests
- gcov

Counters on lines and branches

Update Program as needed

Optimisation
- **Constant forewarding**
- **Inlining**
- **Auto-vectorization**
- **Auto-parallelization**

Results have to remain identical

Compiler intermediate representation

Laboratoire d'Annecy de Physique des Particules

Program

```
float square(float x){
»       return x*x;
}
int main(){
»       float y(42.0f);
»       std::cout << square(y) << std::endl;
»       return 0;
}
```

Graph

Check computing instabilities

Profiler
- GProf

Add probes

Unit Tests
- gcov

Counters on lines and branches

Update Program as needed

Optimisation
- **Constant forewarding**
- **Inlining**
- **Auto-vectorization**
- **Auto-parallelization**

Results have to remain identical

Need to keep a **similar** graph

Compiler intermediate representation

Laboratoire d'Annecy de Physique des Particules

Program

```
float square(float x){
»       return x*x;
}
int main(){
»       float y(42.0f);
»       std::cout << square(y) << std::endl;
»       return 0;
}
```

Graph

Check computing instabilities

Profiler
- GProf

Add probes

Unit Tests
- gcov

Counters on lines and branches

Update Program as needed

Optimisation
- **Constant forewarding**
- **Inlining**
- **Auto-vectorization**
- **Auto-parallelization**

Results have to remain identical

Need to keep a **similar** graph

Some optimisations are **forbidden** by **Graph Theory**

Compilation options

https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html

Compilation options

https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html

- **-O0** : default
  - Try to reduce compilation time, but **-Og** is better for debugging.

Compilation options

`https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html`

- **-O0** : default
  - Try to reduce compilation time, but **-Og** is better for debugging.
- **-O1**
  - Constant forewarding, remove dead code (never called code)...

Compilation options

https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html

- **-O0** : default
  - Try to reduce compilation time, but **-Og** is better for debugging.
- **-O1**
  - Constant forewarding, remove dead code (never called code)...
- **-O2**
  - Partial function inlining, Assume strict aliasing...

Compilation options

https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html

- **-O0** : default
  - Try to reduce compilation time, but **-Og** is better for debugging.
- **-O1**
  - Constant forewarding, remove dead code (never called code)...
- **-O2**
  - Partial function inlining, Assume strict aliasing...
- **-O3**
  - More function inlining, loop unrolling, partial vectorization...

Compilation options

https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html

- **-O0** : default
  - Try to reduce compilation time, but **-Og** is better for debugging.
- **-O1**
  - Constant forewarding, remove dead code (never called code)...
- **-O2**
  - Partial function inlining, Assume strict aliasing...
- **-O3**
  - More function inlining, loop unrolling, partial vectorization...
- **-Ofast**
  - Disregard strict standards compliance. Enable **-ffast-math**, stack size is hardcoded to 32 768 bytes (borrowed from **gfortran**). **Possibly degrades the computation accuracy.**

Examples

Examples

```
float square(float x){
»       return x*x;
}
int main(){
»       float y(42.0f);
»       std::cout << square(y) << std::endl;
»       return 0;
}
```

Examples

```
float square(float x){
»       return x*x;
}
int main(){
»       float y(42.0f);
»       std::cout << square(y) << std::endl;
»       return 0;
}
```

Constant forewarding

```
float square(float x){
»       return x*x;
}
int main(){
»       std::cout << square(42.0f) << std::endl;
»       return 0;
}
```

Examples

```
float square(float x){
»       return x*x;
}
int main(){
»       float y(42.0f);
»       std::cout << square(y) << std::endl;
»       return 0;
}
```

Constant forewarding

```
float square(float x){
»       return x*x;
}
int main(){
»       std::cout << square(42.0f) << std::endl;
»       return 0;
}
```

Function inlining

```
int main(){
»       std::cout << 42.0f*42.0f << std::endl;
»       return 0;
}
```

Examples

```
float square(float x){
»       return x*x;
}
int main(){
»       float y(42.0f);
»       std::cout << square(y) << std::endl;
»       return 0;
}
```

Constant forewarding

```
float square(float x){
»       return x*x;
}
int main(){
»       std::cout << square(42.0f) << std::endl;
»       return 0;
}
```

Function inlining

```
int main(){
»       std::cout << 42.0f*42.0f << std::endl;
»       return 0;
}
```

Constant expression solving

```
int main(){
»       std::cout << 1764.0f << std::endl;
»       return 0;
}
```

Examples

```
float square(float x){
»       return x*x;
}
int main(){
»       float y(42.0f);
»       std::cout << square(y) << std::endl;
»       return 0;
}
```

Constant forewarding

```
float square(float x){
»       return x*x;
}
int main(){
»       std::cout << square(42.0f) << std::endl;
»       return 0;
}
```

Function inlining

```
int main(){
»       std::cout << 42.0f*42.0f << std::endl;
»       return 0;
}
```

Constant expression solving

```
int main(){
»       std::cout << 1764.0f << std::endl;
»       return 0;
}
```

No more :
- **Function call**
- **Multiplication**
- **Variable declaration**

Loop Unrolling

Loop Unrolling

```
void hadamard(float* tabRes, const float * tabX, const float * tabY, size_t nbElement){
	for(size_t i(0lu); i < nbElement; ++i){
		tabRes[i] = tabX[i]*tabY[i];
	}
}
```

Base

Loop Unrolling

```
void hadamard(float* tabRes, const float * tabX, const float * tabY, size_t nbElement){
	for(size_t i(0lu); i < nbElement; ++i){
		tabRes[i] = tabX[i]*tabY[i];
	}
}
```

Base

```
void hadamard(float* tabRes, const float * tabX, const float * tabY, size_t nbElement){
	for(size_t i(0lu); i < nbElement; i += 2lu){
		tabRes[i] = tabX[i]*tabY[i];
		tabRes[i + 1] = tabX[i + 1]*tabY[i + 1];
	}
}
```

Loop unrolled 2 times

Loop Unrolling

```
void hadamard(float* tabRes, const float * tabX, const float * tabY, size_t nbElement){
	for(size_t i(0lu); i < nbElement; ++i){
		tabRes[i] = tabX[i]*tabY[i];
	}
}
```

Base

```
void hadamard(float* tabRes, const float * tabX, const float * tabY, size_t nbElement){
	for(size_t i(0lu); i < nbElement; i += 2lu){
		tabRes[i] = tabX[i]*tabY[i];
		tabRes[i + 1] = tabX[i + 1]*tabY[i + 1];
	}
}
```

Increase Computing in Iteration

Loop unrolled 2 times

Loop Unrolling

```
void hadamard(float* tabRes, const float * tabX, const float * tabY, size_t nbElement){
	for(size_t i(0lu); i < nbElement; ++i){
		tabRes[i] = tabX[i]*tabY[i];
	}
}
```

Base

```
void hadamard(float* tabRes, const float * tabX, const float * tabY, size_t nbElement){
	for(size_t i(0lu); i < nbElement; i += 2lu){
		tabRes[i] = tabX[i]*tabY[i];
		tabRes[i + 1] = tabX[i + 1]*tabY[i + 1];
	}
}
```

Increase Computing in Iteration

Loop unrolled 2 times

```
void hadamard(float* tabRes, const float * tabX, const float * tabY, size_t nbElement){
	for(size_t i(0lu); i < nbElement; i += 4lu){
		tabRes[i] = tabX[i]*tabY[i];
		tabRes[i + 1] = tabX[i + 1]*tabY[i + 1];
		tabRes[i + 2] = tabX[i + 2]*tabY[i + 2];
		tabRes[i + 3] = tabX[i + 3]*tabY[i + 3];
	}
}
```

Loop unrolled 4 times

Loop Unrolling

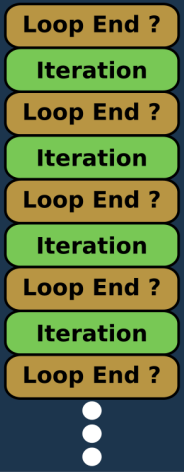
Loop Unrolling

Less end loop check

Loop Unrolling

Less end loop check

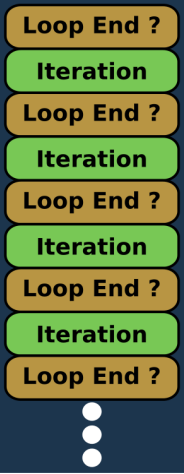
No Loop Unrolling



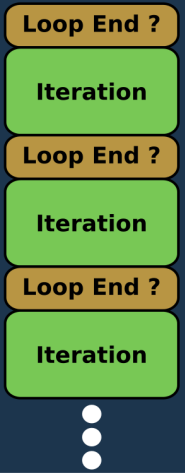
Loop Unrolling

Less end loop check

No Loop Unrolling



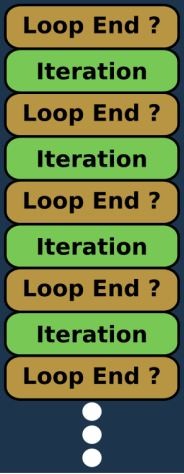
Loop Unrolling 2x



Loop Unrolling

Less end loop check

No Loop Unrolling



Loop Unrolling 2x

Loop Unrolling

Less end loop check

Load / Computing optimisation

No Loop Unrolling

Loop End ?

Iteration

Loop End ?

Iteration

Loop End ?

Iteration

Loop End ?

Iteration

Loop End ?

Loop Unrolling 2x

Loop End ?

Iteration

Loop End ?

Iteration

Loop End ?

Ite

Branching Predicator Stays cool

Loop Unrolling

Less end loop check

No Loop Unrolling

Loop End ?
Iteration
Loop End ?
Iteration
Loop End ?
Iteration
Loop End ?
Iteration
Loop End ?

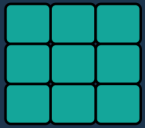
Loop Unrolling 2x

Loop End ?
Iteration
Loop End ?
Iteration
Loop End ?
Ite
Branching Predicator Stays cool

Load / Computing optimisation

No Loop Unrolling

3x3 Kernel Load



Loop Unrolling

Less end loop check

No Loop Unrolling

Loop End ?
Iteration
Loop End ?
Iteration
Loop End ?
Iteration
Loop End ?
Iteration
Loop End ?

Loop Unrolling 2x

Loop End ?
Iteration
Loop End ?
Iteration
Loop End ?
Ite

Branching Predicator Stays cool

Load / Computing optimisation

No Loop Unrolling

3x3 Kernel Load

Loop Unrolling

Less end loop check

No Loop Unrolling

Loop End ?
Iteration
Loop End ?
Iteration
Loop End ?
Iteration
Loop End ?
Iteration
Loop End ?

Loop Unrolling 2x

Loop End ?
Iteration
Loop End ?
Iteration
Loop End ?
Ite

Branching Predicator Stays cool

Load / Computing optimisation

No Loop Unrolling

3x3 Kernel Load

Computing

Store Result

Loop Unrolling

Less end loop check

No Loop Unrolling

Loop End ?
Iteration
Loop End ?
Iteration
Loop End ?
Iteration
Loop End ?
Iteration
Loop End ?

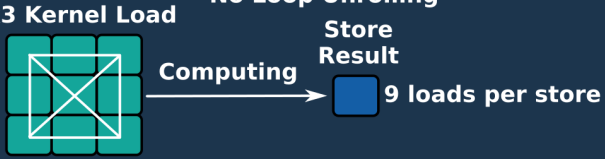
Loop Unrolling 2x

Loop End ?
Iteration
Loop End ?
Iteration
Loop End ?
Ite

Branching Predicator Stays cool

Load / Computing optimisation

No Loop Unrolling



Loop Unrolling

Less end loop check

No Loop Unrolling

Loop End ?
Iteration
Loop End ?
Iteration
Loop End ?
Iteration
Loop End ?
Iteration
Loop End ?

Loop Unrolling 2x

Loop End ?
Iteration
Loop End ?
Iteration
Loop End ?
Ite

Branching Predicator Stays cool

Load / Computing optimisation

No Loop Unrolling

3x3 Kernel Load

Computing

Store Result

9 loads per store

Loop Unrolling 2x

Loop Unrolling

Less end loop check

No Loop Unrolling

- Loop End ?
- Iteration
- Loop End ?
- Iteration
- Loop End ?
- Iteration
- Loop End ?
- Iteration
- Loop End ?

Loop Unrolling 2x

- Loop End ?
- Iteration
- Loop End ?
- Iteration
- Loop End ?
- Ite

Branching Predicator Stays cool

Load / Computing optimisation

No Loop Unrolling

3x3 Kernel Load

Computing

Store Result

9 loads per store

Loop Unrolling 2x

Loop Unrolling

Less end loop check

No Loop Unrolling

Loop End ?
Iteration
Loop End ?
Iteration
Loop End ?
Iteration
Loop End ?
Iteration
Loop End ?

Loop Unrolling 2x

Loop End ?
Iteration
Loop End ?
Iteration
Loop End ?
Ite

Branching Predicator Stays cool

Load / Computing optimisation

No Loop Unrolling

3x3 Kernel Load

Computing

Store Result

9 loads per store

Loop Unrolling 2x

Loop Unrolling

Less end loop check

No Loop Unrolling

Loop End ?
Iteration
Loop End ?
Iteration
Loop End ?
Iteration
Loop End ?
Iteration
Loop End ?

Loop Unrolling 2x

Loop End ?
Iteration
Loop End ?
Iteration
Loop End ?
Ite

Branching Predicator Stays cool

Load / Computing optimisation

No Loop Unrolling

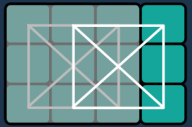
3x3 Kernel Load

Computing

Store Result

9 loads per store

Loop Unrolling 2x



Loop Unrolling

Less end loop check

No Loop Unrolling

Loop End ?
Iteration
Loop End ?
Iteration
Loop End ?
Iteration
Loop End ?
Iteration
Loop End ?

Loop Unrolling 2x

Loop End ?
Iteration
Loop End ?
Iteration
Loop End ?
Ite

Branching Predicator Stays cool

Load / Computing optimisation

No Loop Unrolling

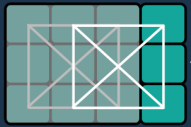
3x3 Kernel Load

Computing

Store Result

9 loads per store

Loop Unrolling 2x



12 loads, 2 stores

Loop Unrolling

Less end loop check

No Loop Unrolling

Loop End ?
Iteration
Loop End ?
Iteration
Loop End ?
Iteration
Loop End ?
Iteration
Loop End ?

Loop Unrolling 2x

Loop End ?
Iteration
Loop End ?
Iteration
Loop End ?
Ite

Branching Predicator Stays cool

Load / Computing optimisation

No Loop Unrolling

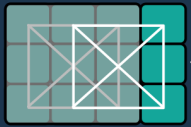
3x3 Kernel Load

Computing

Store Result

9 loads per store

Loop Unrolling 2x



12 loads, 2 stores

6 loads per store

Loop Unrolling

Less end loop check

No Loop Unrolling

Loop End ?
Iteration
Loop End ?
Iteration
Loop End ?
Iteration
Loop End ?
Iteration
Loop End ?

Loop Unrolling 2x

Loop End ?
Iteration
Loop End ?
Iteration
Loop End ?
Ite

Branching Predicator Stays cool

Load / Computing optimisation

No Loop Unrolling

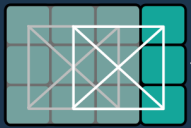
3x3 Kernel Load

Computing

Store Result

9 loads per store

Loop Unrolling 2x



12 loads, 2 stores

6 loads per store

Loop Unrolling 4x

Loop Unrolling

Less end loop check

No Loop Unrolling

Loop End ?
Iteration
Loop End ?
Iteration
Loop End ?
Iteration
Loop End ?
Iteration
Loop End ?

Loop Unrolling 2x

Loop End ?
Iteration
Loop End ?
Iteration
Loop End ?
Ite

Branching Predicator Stays cool

Load / Computing optimisation

No Loop Unrolling

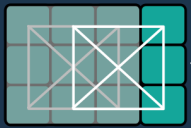
3x3 Kernel Load

Computing

Store Result

9 loads per store

Loop Unrolling 2x



12 loads, 2 stores

6 loads per store

Loop Unrolling 4x

Loop Unrolling

Less end loop check

No Loop Unrolling

Loop End ?
Iteration
Loop End ?
Iteration
Loop End ?
Iteration
Loop End ?
Iteration
Loop End ?

Loop Unrolling 2x

Loop End ?
Iteration
Loop End ?
Iteration
Loop End ?
Ite

Branching Predicator Stays cool

Load / Computing optimisation

No Loop Unrolling

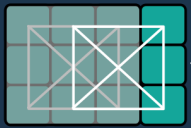
3x3 Kernel Load

Computing

Store Result

9 loads per store

Loop Unrolling 2x



12 loads, 2 stores

6 loads per store

Loop Unrolling 4x

Loop Unrolling

Less end loop check

No Loop Unrolling

Loop End ?
Iteration
Loop End ?
Iteration
Loop End ?
Iteration
Loop End ?
Iteration
Loop End ?

Loop Unrolling 2x

Loop End ?
Iteration
Loop End ?
Iteration
Loop End ?
Ite

Branching Predicator Stays cool

Load / Computing optimisation

No Loop Unrolling

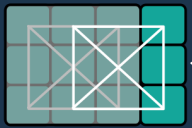
3x3 Kernel Load

Computing

Store Result

9 loads per store

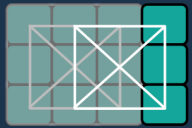
Loop Unrolling 2x



12 loads, 2 stores

6 loads per store

Loop Unrolling 4x



Loop Unrolling

Less end loop check

No Loop Unrolling

Loop End ?
Iteration
Loop End ?
Iteration
Loop End ?
Iteration
Loop End ?
Iteration
Loop End ?

Loop Unrolling 2x

Loop End ?
Iteration
Loop End ?
Iteration
Loop End ?
Ite

Branching Predicator Stays cool

Load / Computing optimisation

No Loop Unrolling

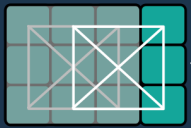
3x3 Kernel Load

Computing

Store Result

9 loads per store

Loop Unrolling 2x



12 loads, 2 stores

6 loads per store

Loop Unrolling 4x

Loop Unrolling

Less end loop check

No Loop Unrolling

Loop End ?
Iteration
Loop End ?
Iteration
Loop End ?
Iteration
Loop End ?
Iteration
Loop End ?

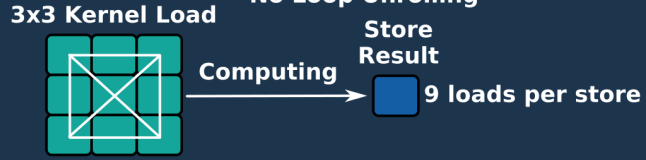
Loop Unrolling 2x

Loop End ?
Iteration
Loop End ?
Iteration
Loop End ?
Ite

Branching Predicator Stays cool

Load / Computing optimisation

No Loop Unrolling



Loop Unrolling 2x

Loop Unrolling 4x

Loop Unrolling

Less end loop check

No Loop Unrolling

Loop End ?
Iteration
Loop End ?
Iteration
Loop End ?
Iteration
Loop End ?
Iteration
Loop End ?

Loop Unrolling 2x

Loop End ?
Iteration
Loop End ?
Iteration
Loop End ?
Ite

Branching Predicator Stays cool

Load / Computing optimisation

No Loop Unrolling

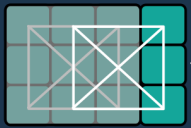
3x3 Kernel Load

Computing

Store Result

9 loads per store

Loop Unrolling 2x



12 loads, 2 stores

6 loads per store

Loop Unrolling 4x

Loop Unrolling

Less end loop check

No Loop Unrolling

Loop End ?
Iteration
Loop End ?
Iteration
Loop End ?
Iteration
Loop End ?
Iteration
Loop End ?

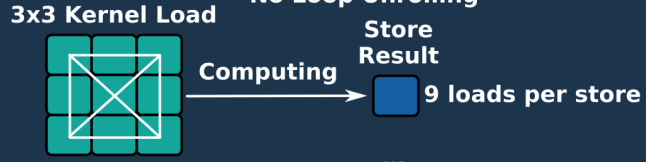
Loop Unrolling 2x

Loop End ?
Iteration
Loop End ?
Iteration
Loop End ?
Ite

Branching Predicator Stays cool

Load / Computing optimisation

No Loop Unrolling



Loop Unrolling

Less end loop check

No Loop Unrolling

Loop End ?
Iteration
Loop End ?
Iteration
Loop End ?
Iteration
Loop End ?
Iteration
Loop End ?

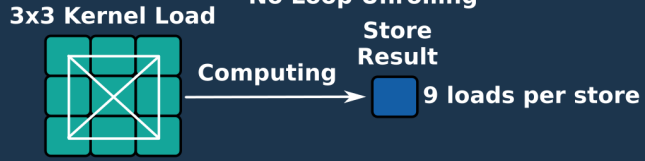
Loop Unrolling 2x

Loop End ?
Iteration
Loop End ?
Iteration
Loop End ?
Ite

Branching Predicator Stays cool

Load / Computing optimisation

No Loop Unrolling



Loop Unrolling

Less end loop check

No Loop Unrolling

Loop End ?
Iteration
Loop End ?
Iteration
Loop End ?
Iteration
Loop End ?
Iteration
Loop End ?

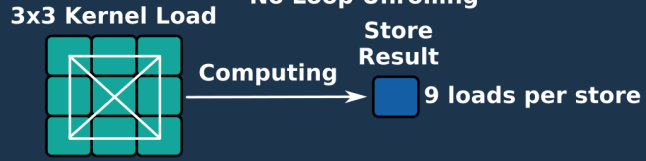
Loop Unrolling 2x

Loop End ?
Iteration
Loop End ?
Iteration
Loop End ?
Ite

Branching Predicator Stays cool

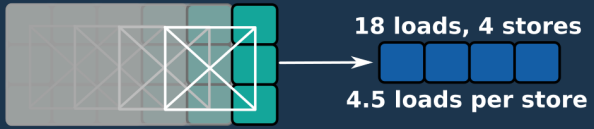
Load / Computing optimisation

No Loop Unrolling



Loop Unrolling 2x

Loop Unrolling 4x



Loop Unrolling

Less end loop check

No Loop Unrolling

Loop End ?
Iteration
Loop End ?
Iteration
Loop End ?
Iteration
Loop End ?
Iteration
Loop End ?

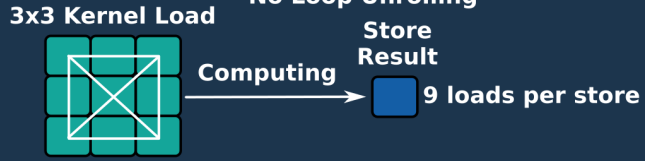
Loop Unrolling 2x

Loop End ?
Iteration
Loop End ?
Iteration
Loop End ?
Ite

Branching Predicator Stays cool

Load / Computing optimisation

No Loop Unrolling

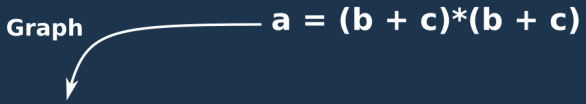


Then it is a problem of number of available registers

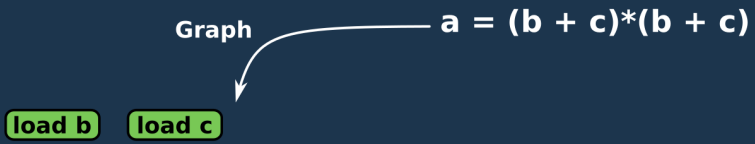
Common subexpression elimination

a = (b + c)*(b + c)

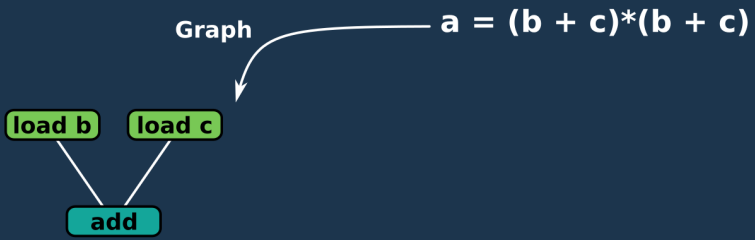
Common subexpression elimination



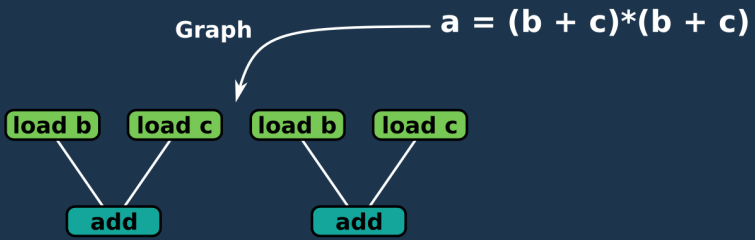
Common subexpression elimination



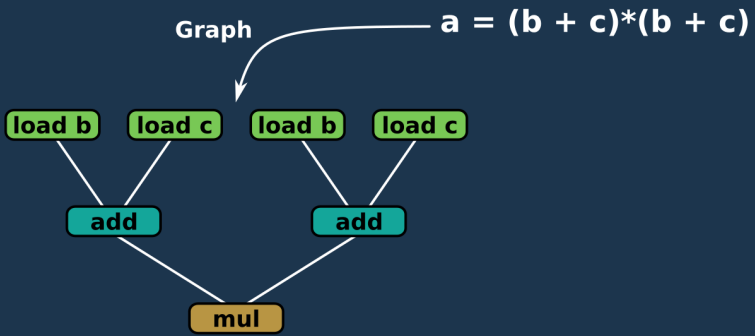
Common subexpression elimination



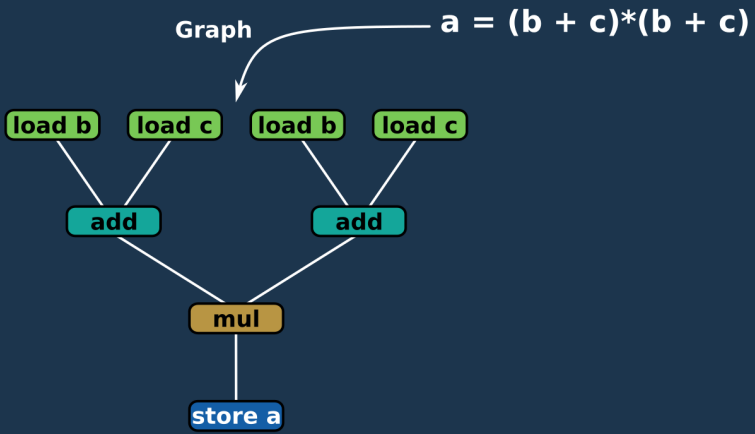
Common subexpression elimination



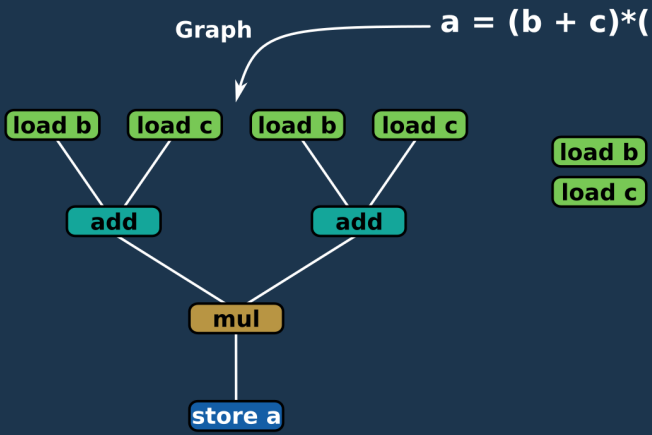
Common subexpression elimination



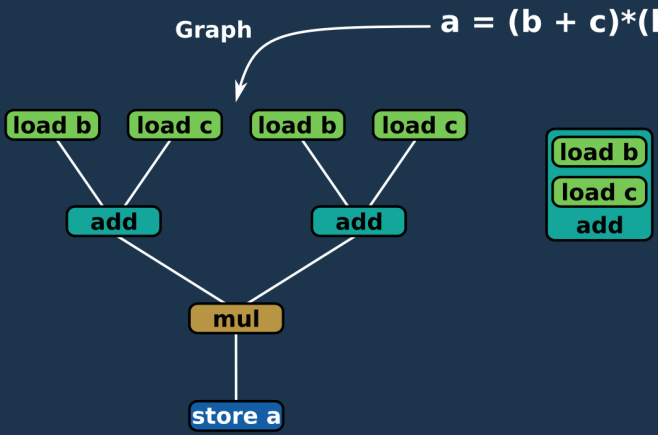
Common subexpression elimination



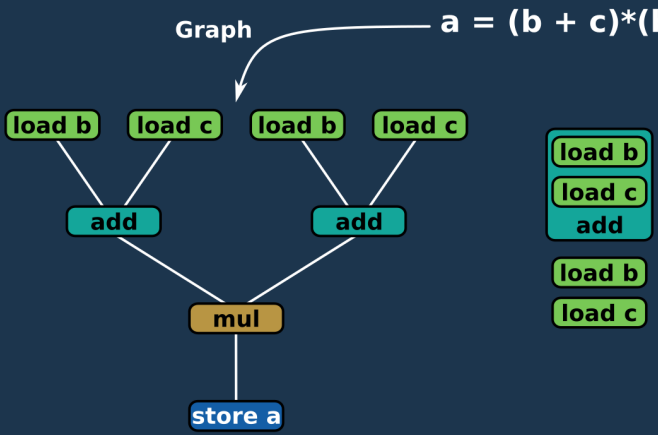
Common subexpression elimination



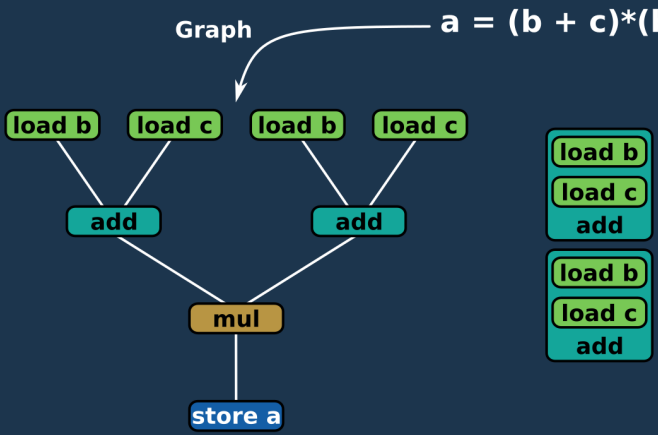
Common subexpression elimination



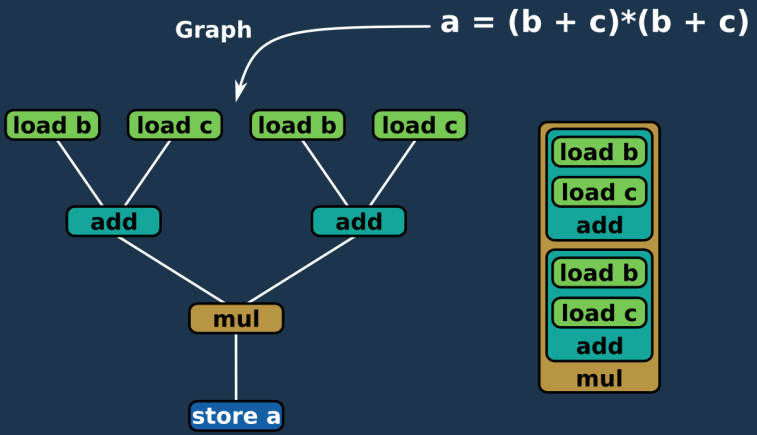
Common subexpression elimination



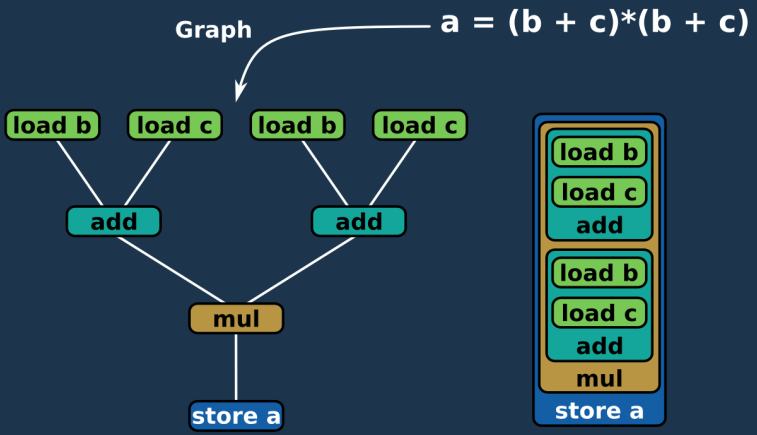
Common subexpression elimination



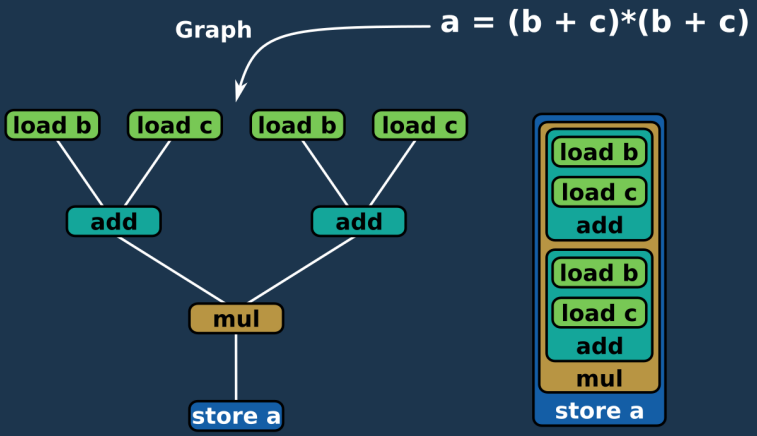
Common subexpression elimination



Common subexpression elimination

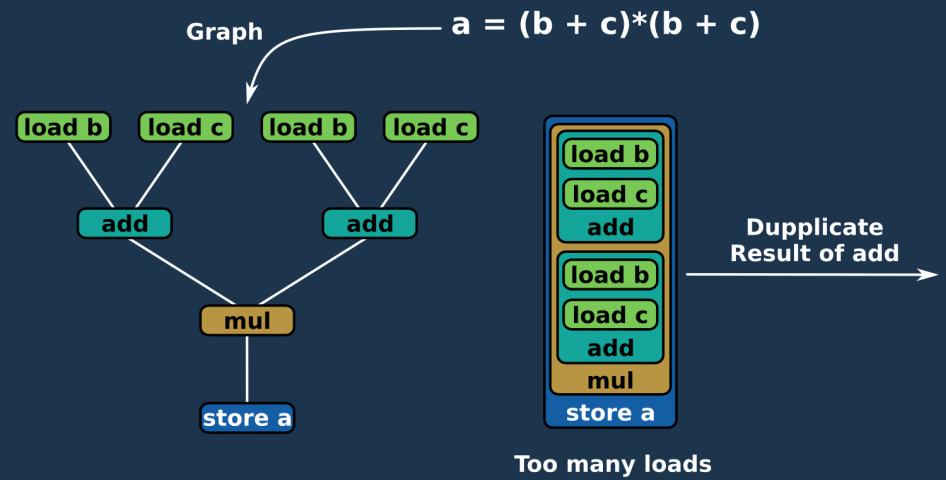


Common subexpression elimination

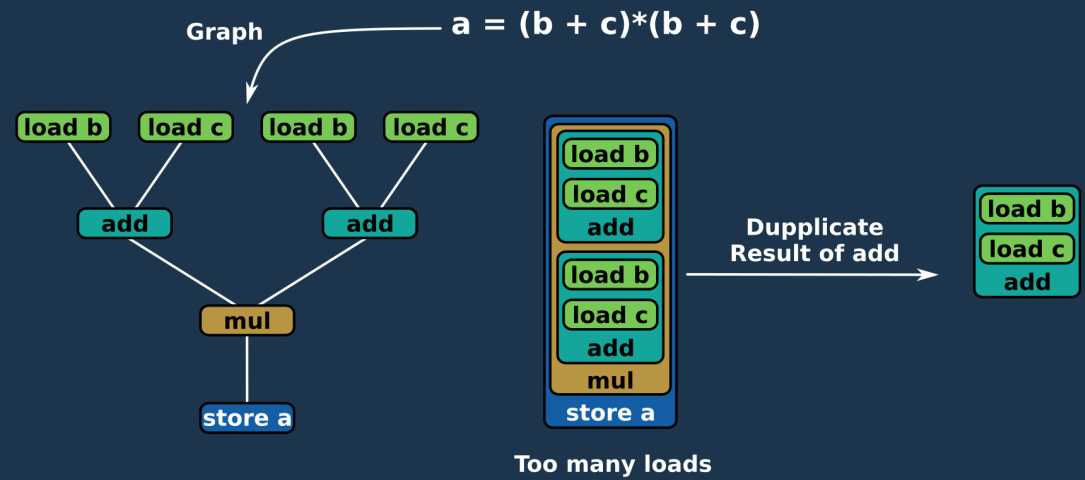


Too many loads

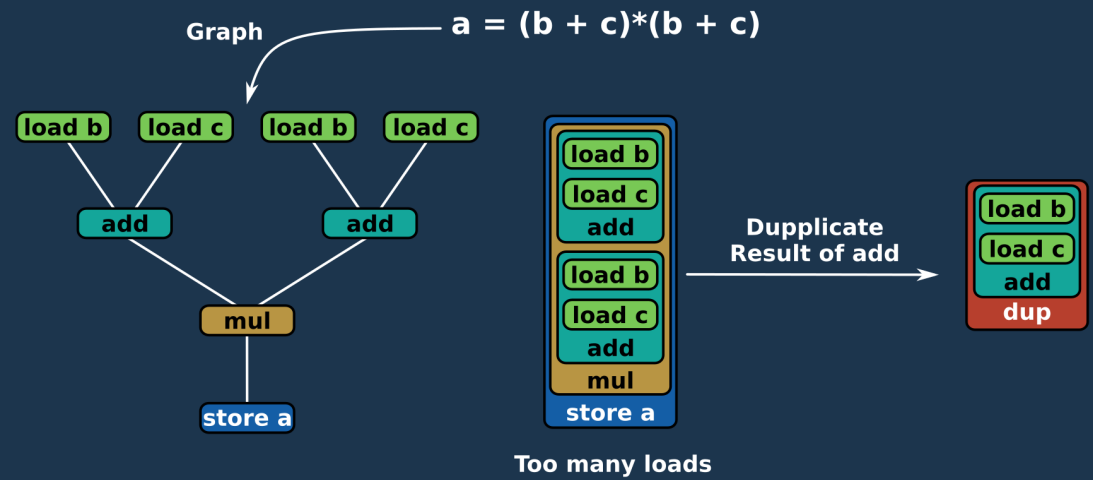
Common subexpression elimination



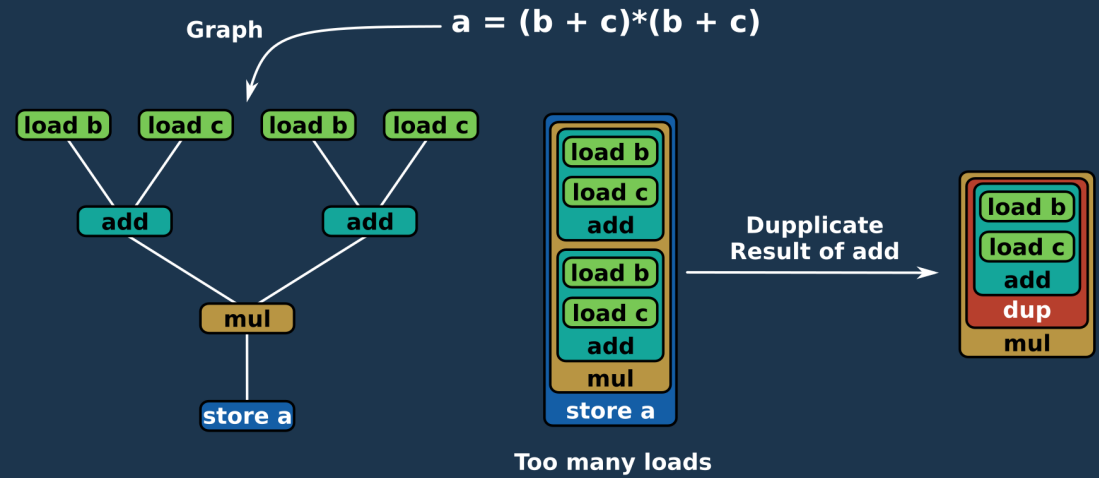
Common subexpression elimination



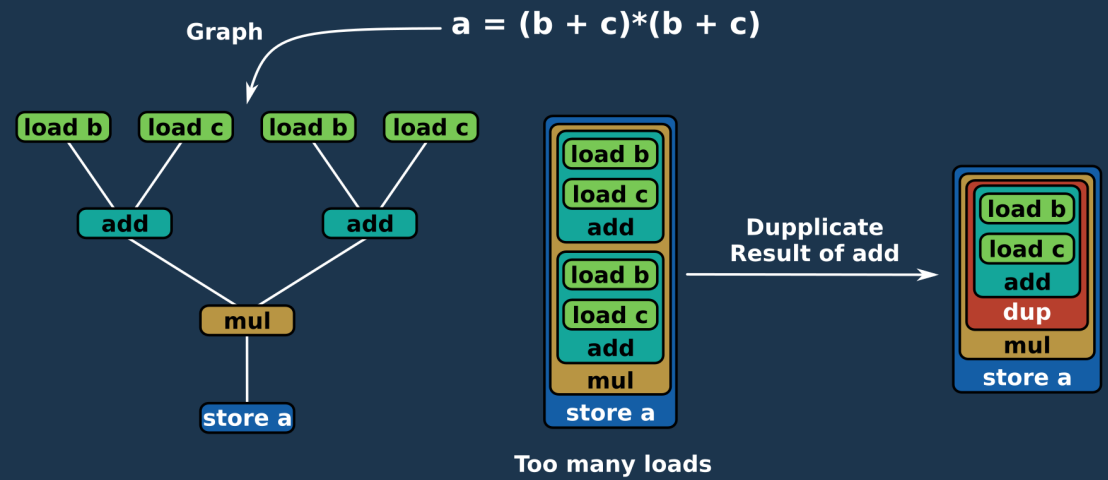
Common subexpression elimination



Common subexpression elimination



Common subexpression elimination



Implementation

Implementation

```
void hadamard_product(float * vecRes, const float * vecX, const float * vecY, size_t nbElement){
»       for(size_t i(0lu); i < nbElement; ++i){
»       »       vecRes[i] = vecX[i]*vecY[i];
»       }
}
```

Implementation

```
void evaluateHadamardProduct(size_t nbElement, size_t nbRepetition){
	//{On déclare nos tableaux et on les initialise :
	float* vecX = (float*)std::malloc(sizeof(float)*nbElement);
	float *vecY = (float*)std::malloc(sizeof(float)*nbElement);
	float *vecRes = (float*)std::malloc(sizeof(float)*nbElement);

	for(size_t i(0lu); i < nbElement; ++i){
		vecX[i] = i*32lu%17lu;
		vecY[i] = i*57lu%31lu;
	}
	size_t fullNbElement(nbElement*nbRepetition);
	//Stating the timer
	HiPeTime beginTime = phoenix_getTime();
	//Let's call the kernel several times
	for(size_t i(0lu); i < nbRepetition; ++i){
		hadamard_product(vecRes, vecX, vecY, nbElement);
	}
	//Stop the timer
	NanoSecs elapsedTime(phoenix_getTime() - beginTime);
	double ellapsedTimeNs(elapsedTime.count()/((double)nbRepetition));
	double timePerElement = ellapsedTimeNs/((double)nbElement), timeErrorPerElement(0.0), ellapsedTimeErrorNs(0.0);

	std::cout << "Hadamard product naive ptr : nbElement = "<<fullNbElement<<", timePerElement = " << timePerElement << " ns/el ±
	"<<timeErrorPerElement<<", elapsedTime = " << ellapsedTimeNs << " ns ± "<<ellapsedTimeErrorNs << std::endl;
	std::cerr << nbElement << "\t" << timePerElement << "\t" << ellapsedTimeNs << "\t" << timeErrorPerElement << "\t" <<
	ellapsedTimeErrorNs << std::endl;

	std::free(vecX);
	std::free(vecY);
	std::free(vecRes);
}
```

Implementation

```
void evaluateHadamardProduct(size_t nbElement, size_t nbRepetition){
	//{On déclare nos tableaux et on les initialise :
	float* vecX = (float*)std::malloc(sizeof(float)*nbElement);
	float *vecY = (float*)std::malloc(sizeof(float)*nbElement);
	float *vecRes = (float*)std::malloc(sizeof(float)*nbElement);

	for(size_t i(0lu); i < nbElement; ++i){
		vecX[i] = i*32lu%17lu;
		vecY[i] = i*57lu%31lu;
	}
	size_t fullNbElement(nbElement*nbRepetition);
	//Stating the timer
	HiPeTime beginTime = phoenix_getTime();
	//Let's call the kernel several times
	for(size_t i(0lu); i < nbRepetition; ++i){
		hadamard_product(vecRes, vecX, vecY, nbElement);
	}
	//Stop the timer
	NanoSecs elapsedTime(phoenix_getTime() - beginTime);
	double ellapsedTimeNs(elapsedTime.count()/((double)nbRepetition));
	double timePerElement = ellapsedTimeNs/((double)nbElement), timeErrorPerElement(0.0), ellapsedTimeErrorNs(0.0);

	std::cout << "Hadamard product naive ptr : nbElement = "<<fullNbElement<<", timePerElement = " << timePerElement << " ns/el ±
	"<<timeErrorPerElement<<", elapsedTime = " << ellapsedTimeNs << " ns ± "<<ellapsedTimeErrorNs << std::endl;
	std::cerr << nbElement << "\t" << timePerElement << "\t" << ellapsedTimeNs << "\t" << timeErrorPerElement << "\t" <<
	ellapsedTimeErrorNs << std::endl;

	std::free(vecX);
	std::free(vecY);
	std::free(vecRes);
}
```

Allocation

Implementation

```
void evaluateHadamardProduct(size_t nbElement, size_t nbRepetition){
	//{On déclare nos tableaux et on les initialise :
	float* vecX = (float*)std::malloc(sizeof(float)*nbElement);
	float *vecY = (float*)std::malloc(sizeof(float)*nbElement);
	float *vecRes = (float*)std::malloc(sizeof(float)*nbElement);

	for(size_t i(0lu); i < nbElement; ++i){
		vecX[i] = i*32lu%17lu;
		vecY[i] = i*57lu%31lu;
	}
	size_t fullNbElement(nbElement*nbRepetition);
	//Stating the timer
	HiPeTime beginTime = phoenix_getTime();
	//Let's call the kernel several times
	for(size_t i(0lu); i < nbRepetition; ++i){
		hadamard_product(vecRes, vecX, vecY, nbElement);
	}
	//Stop the timer
	NanoSecs elapsedTime(phoenix_getTime() - beginTime);
	double ellapsedTimeNs(elapsedTime.count()/((double)nbRepetition));
	double timePerElement = ellapsedTimeNs/((double)nbElement), timeErrorPerElement(0.0), ellapsedTimeErrorNs(0.0);

	std::cout << "Hadamard product naive ptr : nbElement = "<<fullNbElement<<", timePerElement = " << timePerElement << " ns/el ±
	"<<timeErrorPerElement<<", elapsedTime = " << ellapsedTimeNs << " ns ± "<<ellapsedTimeErrorNs << std::endl;
	std::cerr << nbElement << "\t" << timePerElement << "\t" << ellapsedTimeNs << "\t" << timeErrorPerElement << "\t" <<
	ellapsedTimeErrorNs << std::endl;

	std::free(vecX);
	std::free(vecY);
	std::free(vecRes);
}
```

Allocation

Initialisation

Implementation

```
void evaluateHadamardProduct(size_t nbElement, size_t nbRepetition){
	//{On déclare nos tableaux et on les initialise :
	float* vecX = (float*)std::malloc(sizeof(float)*nbElement);
	float *vecY = (float*)std::malloc(sizeof(float)*nbElement);
	float *vecRes = (float*)std::malloc(sizeof(float)*nbElement);

	for(size_t i(0lu); i < nbElement; ++i){
		vecX[i] = i*32lu%17lu;
		vecY[i] = i*57lu%31lu;
	}
	size_t fullNbElement(nbElement*nbRepetition);
	//Stating the timer
	HiPeTime beginTime = phoenix_getTime();
	//Let's call the kernel several times
	for(size_t i(0lu); i < nbRepetition; ++i){
		hadamard_product(vecRes, vecX, vecY, nbElement);
	}
	//Stop the timer
	NanoSecs elapsedTime(phoenix_getTime() - beginTime);
	double ellapsedTimeNs(elapsedTime.count()/((double)nbRepetition));
	double timePerElement = ellapsedTimeNs/((double)nbElement), timeErrorPerElement(0.0), ellapsedTimeErrorNs(0.0);

	std::cout << "Hadamard product naive ptr : nbElement = "<<fullNbElement<<", timePerElement = " << timePerElement << " ns/el ±
	"<<timeErrorPerElement<<", elapsedTime = " << ellapsedTimeNs << " ns ± "<<ellapsedTimeErrorNs << std::endl;
	std::cerr << nbElement << "\t" << timePerElement << "\t" << ellapsedTimeNs << "\t" << timeErrorPerElement << "\t" <<
	ellapsedTimeErrorNs << std::endl;

	std::free(vecX);
	std::free(vecY);
	std::free(vecRes);
}
```

Allocation

Initialisation

Start Chrono

Implementation

```
void evaluateHadamardProduct(size_t nbElement, size_t nbRepetition){
	//{On déclare nos tableaux et on les initialise :
	float* vecX = (float*)std::malloc(sizeof(float)*nbElement);
	float *vecY = (float*)std::malloc(sizeof(float)*nbElement);
	float *vecRes = (float*)std::malloc(sizeof(float)*nbElement);

	for(size_t i(0lu); i < nbElement; ++i){
		vecX[i] = i*32lu%17lu;
		vecY[i] = i*57lu%31lu;
	}
	size_t fullNbElement(nbElement*nbRepetition);
	//Stating the timer
	HiPeTime beginTime = phoenix_getTime();
	//Let's call the kernel several times
	for(size_t i(0lu); i < nbRepetition; ++i){
		hadamard_product(vecRes, vecX, vecY, nbElement);
	}
	//Stop the timer
	NanoSecs elapsedTime(phoenix_getTime() - beginTime);
	double ellapsedTimeNs(elapsedTime.count()/((double)nbRepetition));
	double timePerElement = ellapsedTimeNs/((double)nbElement), timeErrorPerElement(0.0), ellapsedTimeErrorNs(0.0);

	std::cout << "Hadamard product naive ptr : nbElement = "<<fullNbElement<<", timePerElement = " << timePerElement << " ns/el ±
	"<<timeErrorPerElement<<", elapsedTime = " << ellapsedTimeNs << " ns ± "<<ellapsedTimeErrorNs << std::endl;
	std::cerr << nbElement << "\t" << timePerElement << "\t" << ellapsedTimeNs << "\t" << timeErrorPerElement << "\t" <<
	ellapsedTimeErrorNs << std::endl;

	std::free(vecX);
	std::free(vecY);
	std::free(vecRes);
}
```

Allocation

Initialisation

Start Chrono

Performance Evaluation

Implementation

```
void evaluateHadamardProduct(size_t nbElement, size_t nbRepetition){
	//{On déclare nos tableaux et on les initialise :
	float* vecX = (float*)std::malloc(sizeof(float)*nbElement);
	float *vecY = (float*)std::malloc(sizeof(float)*nbElement);
	float *vecRes = (float*)std::malloc(sizeof(float)*nbElement);

	for(size_t i(0lu); i < nbElement; ++i){
		vecX[i] = i*32lu%17lu;
		vecY[i] = i*57lu%31lu;
	}
	size_t fullNbElement(nbElement*nbRepetition);
	//Stating the timer
	HiPeTime beginTime = phoenix_getTime();
	//Let's call the kernel several times
	for(size_t i(0lu); i < nbRepetition; ++i){
		hadamard_product(vecRes, vecX, vecY, nbElement);
	}
	//Stop the timer
	NanoSecs elapsedTime(phoenix_getTime() - beginTime);
	double ellapsedTimeNs(elapsedTime.count()/((double)nbRepetition));
	double timePerElement = ellapsedTimeNs/((double)nbElement), timeErrorPerElement(0.0), ellapsedTimeErrorNs(0.0);

	std::cout << "Hadamard product naive ptr : nbElement = "<<fullNbElement<<", timePerElement = " << timePerElement << " ns/el ±
	"<<timeErrorPerElement<<", elapsedTime = " << ellapsedTimeNs << " ns ± "<<ellapsedTimeErrorNs << std::endl;
	std::cerr << nbElement << "\t" << timePerElement << "\t" << ellapsedTimeNs << "\t" << timeErrorPerElement << "\t" <<
	ellapsedTimeErrorNs << std::endl;

	std::free(vecX);
	std::free(vecY);
	std::free(vecRes);
}
```

Allocation

Initialisation

Start Chrono

Performance Evaluation

Stop Chrono

Implementation

```
void evaluateHadamardProduct(size_t nbElement, size_t nbRepetition){
	//{On déclare nos tableaux et on les initialise :
	float* vecX = (float*)std::malloc(sizeof(float)*nbElement);
	float *vecY = (float*)std::malloc(sizeof(float)*nbElement);
	float *vecRes = (float*)std::malloc(sizeof(float)*nbElement);

	for(size_t i(0lu); i < nbElement; ++i){
		vecX[i] = i*32lu%17lu;
		vecY[i] = i*57lu%31lu;
	}
	size_t fullNbElement(nbElement*nbRepetition);
	//Stating the timer
	HiPeTime beginTime = phoenix_getTime();
	//Let's call the kernel several times
	for(size_t i(0lu); i < nbRepetition; ++i){
		hadamard_product(vecRes, vecX, vecY, nbElement);
	}
	//Stop the timer
	NanoSecs elapsedTime(phoenix_getTime() - beginTime);
	double ellapsedTimeNs(elapsedTime.count()/((double)nbRepetition));
	double timePerElement = ellapsedTimeNs/((double)nbElement), timeErrorPerElement(0.0), ellapsedTimeErrorNs(0.0);

	std::cout << "Hadamard product naive ptr : nbElement = "<<fullNbElement<<", timePerElement = " << timePerElement << " ns/el ±
	"<<timeErrorPerElement<<", elapsedTime = " << ellapsedTimeNs << " ns ± "<<ellapsedTimeErrorNs << std::endl;
	std::cerr << nbElement << "\t" << timePerElement << "\t" << ellapsedTimeNs << "\t" << timeErrorPerElement << "\t" <<
	ellapsedTimeErrorNs << std::endl;

	std::free(vecX);
	std::free(vecY);
	std::free(vecRes);
}
```

Allocation

Initialisation

Start Chrono

Performance Evaluation

Stop Chrono

Time Computation

Implementation

```
void evaluateHadamardProduct(size_t nbElement, size_t nbRepetition){
	//{On déclare nos tableaux et on les initialise :
	float* vecX = (float*)std::malloc(sizeof(float)*nbElement);
	float *vecY = (float*)std::malloc(sizeof(float)*nbElement);
	float *vecRes = (float*)std::malloc(sizeof(float)*nbElement);

	for(size_t i(0lu); i < nbElement; ++i){
		vecX[i] = i*32lu%17lu;
		vecY[i] = i*57lu%31lu;
	}
	size_t fullNbElement(nbElement*nbRepetition);
	//Stating the timer
	HiPeTime beginTime = phoenix_getTime();
	//Let's call the kernel several times
	for(size_t i(0lu); i < nbRepetition; ++i){
		hadamard_product(vecRes, vecX, vecY, nbElement);
	}
	//Stop the timer
	NanoSecs elapsedTime(phoenix_getTime() - beginTime);
	double ellapsedTimeNs(elapsedTime.count()/((double)nbRepetition));
	double timePerElement = ellapsedTimeNs/((double)nbElement), timeErrorPerElement(0.0), ellapsedTimeErrorNs(0.0);

	std::cout << "Hadamard product naive ptr : nbElement = "<<fullNbElement<<", timePerElement = " << timePerElement << " ns/el ±
	"<<timeErrorPerElement<<", elapsedTime = " << ellapsedTimeNs << " ns ± "<<ellapsedTimeErrorNs << std::endl;
	std::cerr << nbElement << "\t" << timePerElement << "\t" << ellapsedTimeNs << "\t" << timeErrorPerElement << "\t" <<
	ellapsedTimeErrorNs << std::endl;

	std::free(vecX);
	std::free(vecY);
	std::free(vecRes);
}
```

Allocation

Initialisation

Start Chrono

Performance Evaluation

Stop Chrono

Time Computation

Print Results

Implementation

```
void evaluateHadamardProduct(size_t nbElement, size_t nbRepetition){
	//{On déclare nos tableaux et on les initialise :
	float* vecX = (float*)std::malloc(sizeof(float)*nbElement);
	float *vecY = (float*)std::malloc(sizeof(float)*nbElement);
	float *vecRes = (float*)std::malloc(sizeof(float)*nbElement);

	for(size_t i(0lu); i < nbElement; ++i){
		vecX[i] = i*32lu%17lu;
		vecY[i] = i*57lu%31lu;
	}
	size_t fullNbElement(nbElement*nbRepetition);
	//Stating the timer
	HiPeTime beginTime = phoenix_getTime();
	//Let's call the kernel several times
	for(size_t i(0lu); i < nbRepetition; ++i){
		hadamard_product(vecRes, vecX, vecY, nbElement);
	}
	//Stop the timer
	NanoSecs elapsedTime(phoenix_getTime() - beginTime);
	double ellapsedTimeNs(elapsedTime.count()/((double)nbRepetition));
	double timePerElement = ellapsedTimeNs/((double)nbElement), timeErrorPerElement(0.0), ellapsedTimeErrorNs(0.0);

	std::cout << "Hadamard product naive ptr : nbElement = "<<fullNbElement<<", timePerElement = " << timePerElement << " ns/el ±
	"<<timeErrorPerElement<<", elapsedTime = " << ellapsedTimeNs << " ns ± "<<ellapsedTimeErrorNs << std::endl;
	std::cerr << nbElement << "\t" << timePerElement << "\t" << ellapsedTimeNs << "\t" << timeErrorPerElement << "\t" <<
	ellapsedTimeErrorNs << std::endl;

	std::free(vecX);
	std::free(vecY);
	std::free(vecRes);
}
```

Allocation

Initialisation

Start Chrono

Performance Evaluation

Stop Chrono

Time Computation

Print Results

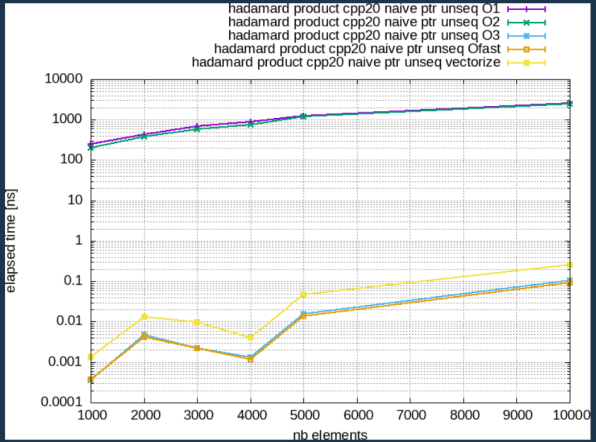
Deallocation

Implementation

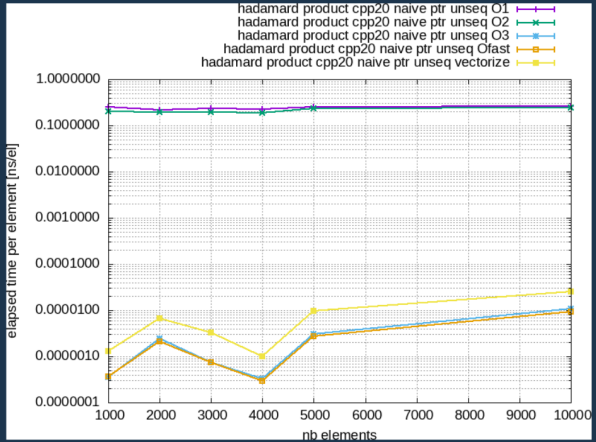
```
int main(int argc, char** argv){
»       evaluateHadamardProduct(1000lu, 1000000lu);
»       evaluateHadamardProduct(2000lu, 500000lu);
»       evaluateHadamardProduct(3000lu, 300000lu);
»       evaluateHadamardProduct(4000lu, 2500000lu);
»       evaluateHadamardProduct(5000lu, 200000lu);
»       evaluateHadamardProduct(10000lu, 100000lu);
»       return 0;
}
```

Performance G++11

Total time

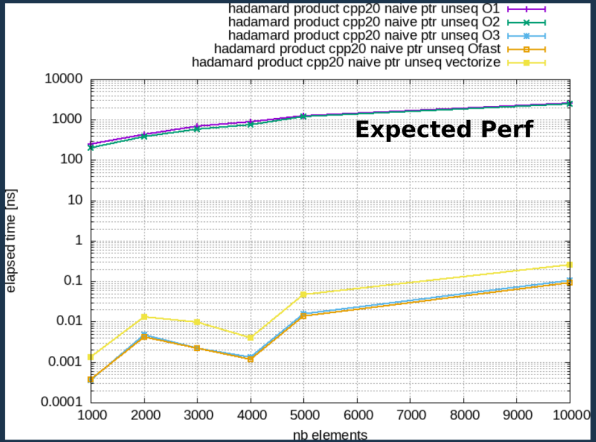


Time per element

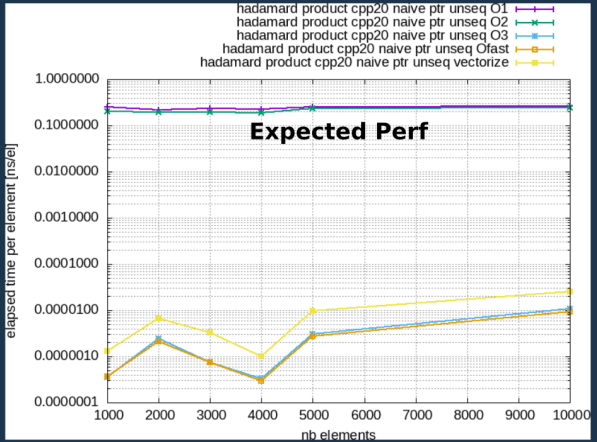


Performance G++11

Total time

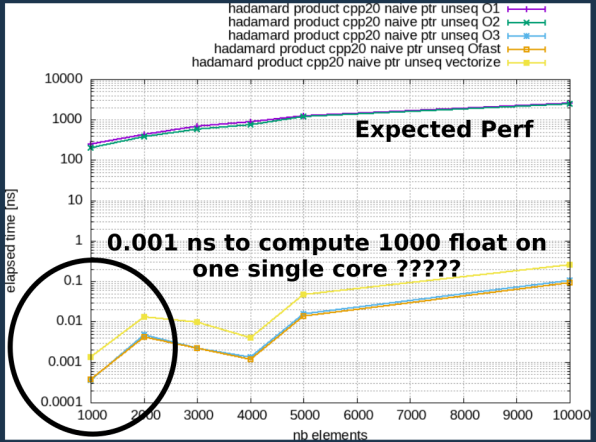


Time per element

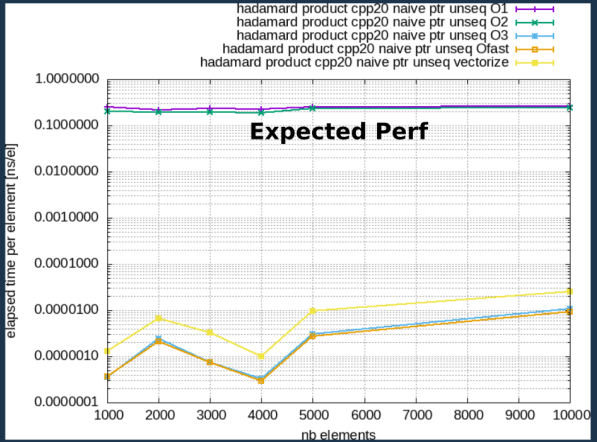


Performance G++11

Total time

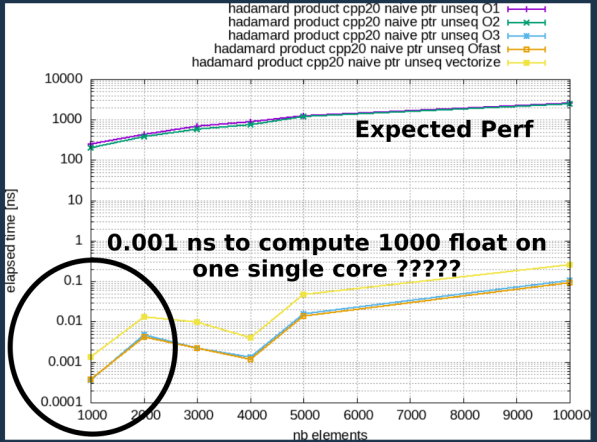


Time per element

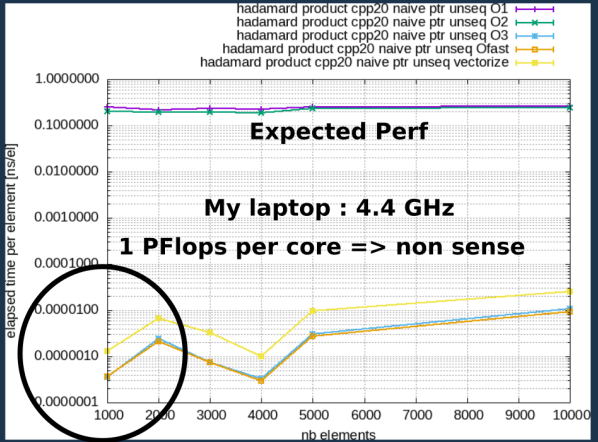


Performance G++11

Total time

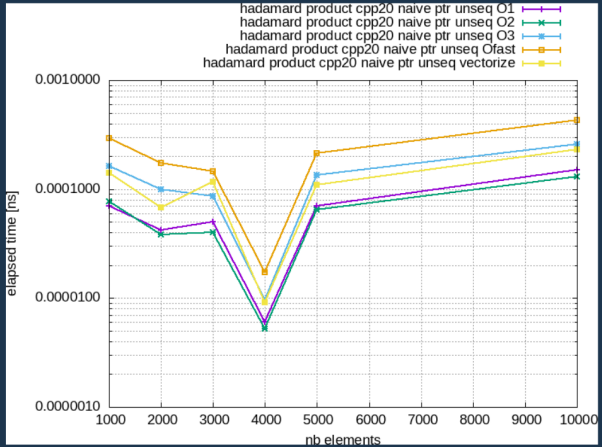


Time per element

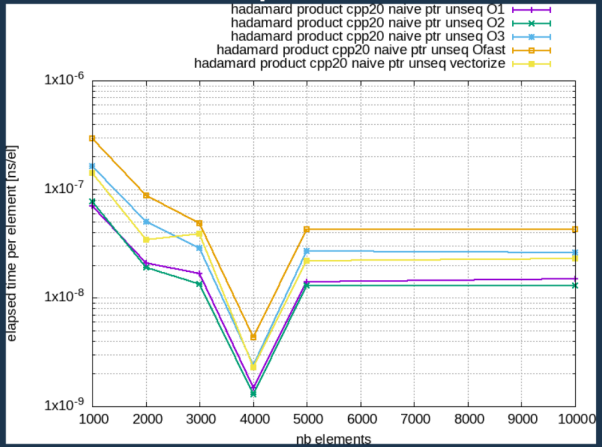


Performance Clang++14

Total time

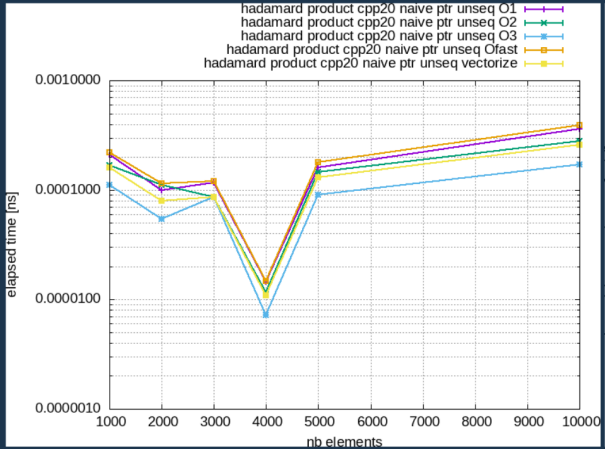


Time per element

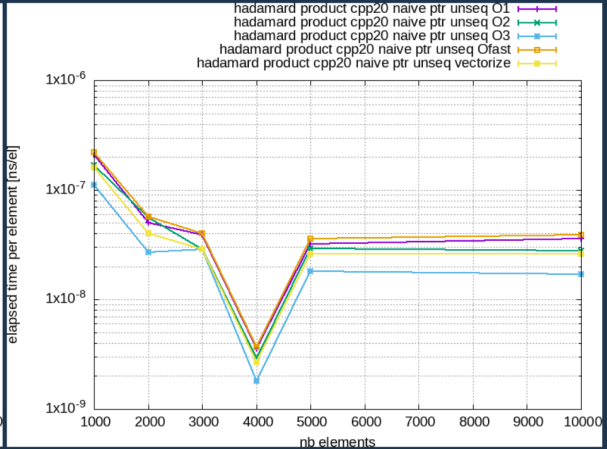


Performance ICPX 2020 00 02

Total time



Time per element



Implementation + Compiler

```
void evaluateHadamardProduct(size_t nbElement, size_t nbRepetition){
	//{On déclare nos tableaux et on les initialise :
	float* vecX = (float*)std::malloc(sizeof(float)*nbElement);
	float *vecY = (float*)std::malloc(sizeof(float)*nbElement);
	float *vecRes = (float*)std::malloc(sizeof(float)*nbElement);

	for(size_t i(0lu); i < nbElement; ++i){
		vecX[i] = i*32lu%17lu;
		vecY[i] = i*57lu%31lu;
	}
	size_t fullNbElement(nbElement*nbRepetition);
	//Stating the timer
	HiPeTime beginTime = phoenix_getTime();
	//Let's call the kernel several times
	for(size_t i(0lu); i < nbRepetition; ++i){
		hadamard_product(vecRes, vecX, vecY, nbElement);
	}
	//Stop the timer
	NanoSecs elapsedTime(phoenix_getTime() - beginTime);
	double ellapsedTimeNs(elapsedTime.count()/((double)nbRepetition));
	double timePerElement = ellapsedTimeNs/((double)nbElement), timeErrorPerElement(0.0), ellapsedTimeErrorNs(0.0);

	std::cout << "Hadamard product naive ptr : nbElement = "<<fullNbElement<<", timePerElement = " << timePerElement << " ns/el ±
	"<<timeErrorPerElement<<", elapsedTime = " << ellapsedTimeNs << " ns ± "<<ellapsedTimeErrorNs << std::endl;
	std::cerr << nbElement << "\t" << timePerElement << "\t" << ellapsedTimeNs << "\t" << timeErrorPerElement << "\t" <<
	ellapsedTimeErrorNs << std::endl;

	std::free(vecX);
	std::free(vecY);
	std::free(vecRes);
}
```

Implementation + Compiler

```
void evaluateHadamardProduct(size_t nbElement, size_t nbRepetition){
	//{On déclare nos tableaux et on les initialise :
	float* vecX = (float*)std::malloc(sizeof(float)*nbElement);
	float* vecY = (float*)std::malloc(sizeof(float)*nbElement);
	float *vecRes = (float*)std::malloc(sizeof(float)*nbElement);

	for(size_t i(0lu); i < nbElement; ++i){
		vecX[i] = i*32lu%17lu;
		vecY[i] = i*57lu%31lu;
	}
	size_t fullNbElement(nbElement*nbRepetition);
	//Stating the timer
	HiPeTime beginTime = phoenix_getTime();
	//Let's call the kernel several times
	for(size_t i(0lu); i < nbRepetition; ++i){
		hadamard_product(vecRes, vecX, vecY, nbElement);
	}
	//Stop the timer
	NanoSecs elapsedTime(phoenix_getTime() - beginTime);
	double ellapsedTimeNs(elapsedTime.count()/((double)nbRepetition));
	double timePerElement = ellapsedTimeNs/((double)nbElement), timeErrorPerElement(0.0), ellapsedTimeErrorNs(0.0);

	std::cout << "Hadamard product naive ptr : nbElement = "<<fullNbElement<<", timePerElement = " << timePerElement << " ns/el ±
	"<<timeErrorPerElement<<", elapsedTime = " << ellapsedTimeNs << " ns ± "<<ellapsedTimeErrorNs << std::endl;
	std::cerr << nbElement << "\t" << timePerElement << "\t" << ellapsedTimeNs << "\t" << timeErrorPerElement << "\t" <<
	ellapsedTimeErrorNs << std::endl;

	std::free(vecX);
	std::free(vecY);
	std::free(vecRes);
}
```

Implementation + Compiler

```
void evaluateHadamardProduct(size_t nbElement, size_t nbRepetition){
	//{On déclare nos tableaux et on les initialise :
	float* vecX = (float*)std::malloc(sizeof(float)*nbElement);
	float* vecY = (float*)std::malloc(sizeof(float)*nbElement);
	float *vecRes = (float*)std::malloc(sizeof(float)*nbElement);

	for(size_t i(0lu); i < nbElement; ++i){
		vecX[i] = i*32lu%17lu;
		vecY[i] = i*57lu%31lu;
	}
	size_t fullNbElement(nbElement*nbRepetition);
	//Stating the timer
	HiPeTime beginTime = phoenix_getTime();
	//Let's call the kernel several times
	for(size_t i(0lu); i < nbRepetition; ++i){
		hadamard_product(vecRes, vecX, vecY, nbElement);
	}
	//Stop the timer
	NanoSecs elapsedTime(phoenix_getTime() - beginTime);
	double ellapsedTimeNs(elapsedTime.count()/((double)nbRepetition));
	double timePerElement = ellapsedTimeNs/((double)nbElement), timeErrorPerElement(0.0), ellapsedTimeErrorNs(0.0);

	std::cout << "Hadamard product naive ptr : nbElement = "<<fullNbElement<<", timePerElement = " << timePerElement << " ns/el ±
	"<<timeErrorPerElement<<", elapsedTime = " << ellapsedTimeNs << " ns ± "<<ellapsedTimeErrorNs << std::endl;
	std::cerr << nbElement << "\t" << timePerElement << "\t" << ellapsedTimeNs << "\t" << timeErrorPerElement << "\t" <<
	ellapsedTimeErrorNs << std::endl;

	std::free(vecX);
	std::free(vecY);
	std::free(vecRes);
}
```

Result of computation is not used

Implementation + Compiler

```
void evaluateHadamardProduct(size_t nbElement, size_t nbRepetition){
	//{On déclare nos tableaux et on les initialise :
	float* vecX = (float*)std::malloc(sizeof(float)*nbElement);
	float *vecY = (float*)std::malloc(sizeof(float)*nbElement);
	float *vecRes = (float*)std::malloc(sizeof(float)*nbElement);

	for(size_t i(0lu); i < nbElement; ++i){
		vecX[i] = i*32lu%17lu;
		vecY[i] = i*57lu%31lu;
	}
	size_t fullNbElement(nbElement*nbRepetition);
	//Stating the timer
	HiPeTime beginTime = phoenix_getTime();
	//Let's call the kernel several times
	for(size_t i(0lu); i < nbRepetition; ++i){
		hadamard_product(vecRes, vecX, vecY, nbElement);
	}
	//Stop the timer
	NanoSecs elapsedTime(phoenix_getTime() - beginTime);
	double ellapsedTimeNs(elapsedTime.count()/((double)nbRepetition));
	double timePerElement = ellapsedTimeNs/((double)nbElement), timeErrorPerElement(0.0), ellapsedTimeErrorNs(0.0);

	std::cout << "Hadamard product naive ptr : nbElement = "<<fullNbElement<<", timePerElement = " << timePerElement << " ns/el ±
	"<<timeErrorPerElement<<", elapsedTime = " << ellapsedTimeNs << " ns ± "<<ellapsedTimeErrorNs << std::endl;
	std::cerr << nbElement << "\t" << timePerElement << "\t" << ellapsedTimeNs << "\t" << timeErrorPerElement << "\t" <<
	ellapsedTimeErrorNs << std::endl;

	std::free(vecX);
	std::free(vecY);
	std::free(vecRes);
}
```

Implementation + Compiler

```
void evaluateHadamardProduct(size_t nbElement, size_t nbRepetition){
	//{On déclare nos tableaux et on les initialise :
	float* vecX = (float*)std::malloc(sizeof(float)*nbElement);
	float *vecY = (float*)std::malloc(sizeof(float)*nbElement);
	float *vecRes = (float*)std::malloc(sizeof(float)*nbElement);

	for(size_t i(0lu); i < nbElement; ++i){
		vecX[i] = i*32lu%17lu;
		vecY[i] = i*57lu%31lu;
	}
	size_t fullNbElement(nbElement*nbRepetition);
	//Stating the timer
	HiPeTime beginTime = phoenix_getTime();
	//Let's call the kernel several times
	for(size_t i(0lu); i < nbRepetition; ++i){
		hadamard_product(vecRes, vecX, vecY, nbElement);
	}
	//Stop the timer
	NanoSecs elapsedTime(phoenix_getTime() - beginTime);
	double ellapsedTimeNs(elapsedTime.count()/((double)nbRepetition));
	double timePerElement = ellapsedTimeNs/((double)nbElement), timeErrorPerElement(0.0), ellapsedTimeErrorNs(0.0);

	std::cout << "Hadamard product naive ptr : nbElement = "<<fullNbElement<<", timePerElement = " << timePerElement << " ns/el ±
	"<<timeErrorPerElement<<", elapsedTime = " << ellapsedTimeNs << " ns ± "<<ellapsedTimeErrorNs << std::endl;
	std::cerr << nbElement << "\t" << timePerElement << "\t" << ellapsedTimeNs << "\t" << timeErrorPerElement << "\t" <<
	ellapsedTimeErrorNs << std::endl;

	std::free(vecX);
	std::free(vecY);
	std::free(vecRes);
}
```

Useless loop

Implementation + Compiler

```
void evaluateHadamardProduct(size_t nbElement, size_t nbRepetition){
	//{On déclare nos tableaux et on les initialise :
	float* vecX = (float*)std::malloc(sizeof(float)*nbElement);
	float *vecY = (float*)std::malloc(sizeof(float)*nbElement);
	float *vecRes = (float*)std::malloc(sizeof(float)*nbElement);

	for(size_t i(0lu); i < nbElement; ++i){
		vecX[i] = i*32lu%17lu;
		vecY[i] = i*57lu%31lu;
	}
	size_t fullNbElement(nbElement*nbRepetition);
	//Stating the timer
	HiPeTime beginTime = phoenix_getTime();
	//Let's call the kernel several times
	for(size_t i(0lu); i < nbRepetition; ++i){
		hadamard_product(vecRes, vecX, vecY, nbElement);
	}
	//Stop the timer
	NanoSecs elapsedTime(phoenix_getTime() - beginTime);
	double ellapsedTimeNs(elapsedTime.count()/((double)nbRepetition));
	double timePerElement = ellapsedTimeNs/((double)nbElement), timeErrorPerElement(0.0), ellapsedTimeErrorNs(0.0);

	std::cout << "Hadamard product naive ptr : nbElement = "<<fullNbElement<<", timePerElement = " << timePerElement << " ns/el ±
	"<<timeErrorPerElement<<", elapsedTime = " << ellapsedTimeNs << " ns ± "<<ellapsedTimeErrorNs << std::endl;
	std::cerr << nbElement << "\t" << timePerElement << "\t" << ellapsedTimeNs << "\t" << timeErrorPerElement << "\t" <<
	ellapsedTimeErrorNs << std::endl;

	std::free(vecX);
	std::free(vecY);
	std::free(vecRes);
}
```

Implementation + Compiler

```
void evaluateHadamardProduct(size_t nbElement, size_t nbRepetition){
    //{On déclare nos tableaux et on les initialise :
    float* vecX = (float*)std::malloc(sizeof(float)*nbElement);
    float *vecY = (float*)std::malloc(sizeof(float)*nbElement);
    float *vecRes = (float *)std::malloc(sizeof(float)*nbElement);

    for(size_t i=0lu; i < nbElement; ++i){
        vecX[i] = i*32lu%17lu;
        vecY[i] = i*57lu%31lu;
    }
    size_t fullNbElement(nbElement*nbRepetition);
    //Stating the timer
    HiPeTime beginTime = phoenix_getTime();
    //Let's call the kernel several times
    for(size_t i=0lu; i < nbRepetition; ++i){
        hadamard_product(vecRes, vecX, vecY, nbElement);
    }
    //Stop the timer
    NanoSecs elapsedTime(phoenix_getTime() - beginTime);
    double ellapsedTimeNs(elapsedTime.count()/((double)nbRepetition));
    double timePerElement = ellapsedTimeNs/((double)nbElement), timeErrorPerElement(0.0), ellapsedTimeErrorNs(0.0);

    std::cout << "Hadamard product naive ptr : nbElement = "<<fullNbElement<<", timePerElement = " << timePerElement << " ns/el ±
    "<<timeErrorPerElement<<", elapsedTime = " << ellapsedTimeNs << " ns ± "<<ellapsedTimeErrorNs << std::endl;
    std::cerr << nbElement << "\t" << timePerElement << "\t" << ellapsedTimeNs << "\t" << timeErrorPerElement << "\t" <<
    ellapsedTimeErrorNs << std::endl;

    std::free(vecX);
    std::free(vecY);
    std::free(vecRes);
}
```

Variables set but not used

Implementation + Compiler

```cpp
void evaluateHadamardProduct(size_t nbElement, size_t nbRepetition){
	//{On déclare nos tableaux et on les initialise :
	float *vecX = (float*)std::malloc(sizeof(float)*nbElement);
	float *vecY = (float*)std::malloc(sizeof(float)*nbElement);
	float *vecRes = (float*)std::malloc(sizeof(float)*nbElement);
	
	for(size_t i(0lu); i < nbElement; ++i){
		vecX[i] = i*32lu%17lu;
		vecY[i] = i*57lu%31lu;
	}
	size_t fullNbElement(nbElement*nbRepetition);
	//Stating the timer
	HiPeTime beginTime = phoenix_getTime();
	//Let's call the kernel several times
	for(size_t i(0lu); i < nbRepetition; ++i){
		hadamard_product(vecRes, vecX, vecY, nbElement);
	}
	//Stop the timer
	NanoSecs elapsedTime(phoenix_getTime() - beginTime);
	double ellapsedTimeNs(elapsedTime.count()/((double)nbRepetition));
	double timePerElement = ellapsedTimeNs/((double)nbElement), timeErrorPerElement(0.0), ellapsedTimeErrorNs(0.0);
	
	std::cout << "Hadamard product naive ptr : nbElement = "<<fullNbElement<<", timePerElement = " << timePerElement << " ns/el ± "<<timeErrorPerElement<<", elapsedTime = " << ellapsedTimeNs << " ns ± "<<ellapsedTimeErrorNs << std::endl;
	std::cerr << nbElement << "\t" << timePerElement << "\t" << ellapsedTimeNs << "\t" << timeErrorPerElement << "\t" << ellapsedTimeErrorNs << std::endl;
	
	std::free(vecX);
	std::free(vecY);
	std::free(vecRes);
}
```

Implementation + Compiler

```
void evaluateHadamardProduct(size_t nbElement, size_t nbRepetition){
        //{On déclare nos tableaux et on les initialise :
        [illegible]
        [illegible]
        [illegible]

        for(size_t i(0lu); i < nbElement; ++i){
                [illegible]
                [illegible]
        }
        size_t fullNbElement(nbElement*nbRepetition);
        //Stating the timer
        HiPeTime beginTime = phoenix_getTime();
        //Let's call the kernel several times
        [illegible]
                [illegible]
        }
        //Stop the timer
        NanoSecs elapsedTime(phoenix_getTime() - beginTime);
        double ellapsedTimeNs(elapsedTime.count()/((double)nbRepetition));
        double timePerElement = ellapsedTimeNs/((double)nbElement), timeErrorPerElement(0.0), ellapsedTimeErrorNs(0.0);

        std::cout << "Hadamard product naive ptr : nbElement = "<<fullNbElement<<", timePerElement = " << timePerElement << " ns/el ±
        "<<timeErrorPerElement<<", elapsedTime = " << ellapsedTimeNs << " ns ± "<<ellapsedTimeErrorNs << std::endl;
        std::cerr << nbElement << "\t" << timePerElement << "\t" << ellapsedTimeNs << "\t" << timeErrorPerElement << "\t" <<
        ellapsedTimeErrorNs << std::endl;

        [illegible]
        [illegible]
        [illegible]
}
```

Useless loop

Implementation + Compiler

```
void evaluateHadamardProduct(size_t nbElement, size_t nbRepetition){
	//{On déclare nos tableaux et on les initialise :
	float* vecX = (float*)std::malloc(sizeof(float)*nbElement);
	float* vecY = (float*)std::malloc(sizeof(float)*nbElement);
	float* vecRes = (float*)std::malloc(sizeof(float)*nbElement);

	for(size_t i=0;i<nbElement;++i){
		vecX[i] = 2.0f*i+1.0f;
		vecY[i] = 2.0f*i+2.0f;
	}
	size_t fullNbElement(nbElement*nbRepetition);
	//Stating the timer
	HiPeTime beginTime = phoenix_getTime();
	//Let's call the kernel several times
	for(size_t i=0;i<nbRepetition;++i){
		hadamard_product(vecRes, vecX, vecY, nbElement);
	}
	//Stop the timer
	NanoSecs elapsedTime(phoenix_getTime() - beginTime);
	double ellapsedTimeNs(elapsedTime.count()/((double)nbRepetition));
	double timePerElement = ellapsedTimeNs/((double)nbElement), timeErrorPerElement(0.0), ellapsedTimeErrorNs(0.0);

	std::cout << "Hadamard product naive ptr : nbElement = "<<fullNbElement<<", timePerElement = " << timePerElement << " ns/el ±
	"<<timeErrorPerElement<<", elapsedTime = " << ellapsedTimeNs << " ns ± "<<ellapsedTimeErrorNs << std::endl;
	std::cerr << nbElement << "\t" << timePerElement << "\t" << ellapsedTimeNs << "\t" << timeErrorPerElement << "\t" <<
	ellapsedTimeErrorNs << std::endl;

	std::free(vecX);
	std::free(vecY);
	std::free(vecRes);
}
```

Implementation + Compiler

```
void evaluateHadamardProduct(size_t nbElement, size_t nbRepetition){
      //{On déclare nos tableaux et on les initialise :
      float* vecX = (float*) std::malloc(sizeof(float)*nbElement);
      float* vecY = (float*) std::malloc(sizeof(float)*nbElement);
      float* vecRes = (float*) std::malloc(sizeof(float)*nbElement);

      for(size_t i=0;i<nbElement;++i){
            vecX[i] = 2.0f*i+1.0f;
            vecY[i] = 2.0f*i-1.0f;
      }
      size_t fullNbElement(nbElement*nbRepetition);
      //Stating the timer
      HiPeTime beginTime = phoenix_getTime();
      //Let's call the kernel several times
      for(size_t i=0;i<nbRepetition;++i){
            hadamard_product(vecRes, vecX, vecY, nbElement);
      }
      //Stop the timer
      NanoSecs elapsedTime(phoenix_getTime() - beginTime);
      double ellapsedTimeNs(elapsedTime.count()/((double)nbRepetition));
      double timePerElement = ellapsedTimeNs/((double)nbElement), timeErrorPerElement(0.0), ellapsedTimeErrorNs(0.0);
      std::cout << "Hadamard product naive ptr : nbElement = "<<fullNbElement<<", timePerElement = " << timePerElement << " ns/el ±
      "<<timeErrorPerElement<<", elapsedTime = " << ellapsedTimeNs << " ns ± "<<ellapsedTimeErrorNs << std::endl;
      std::cerr << nbElement << "\t" << timePerElement << "\t" << ellapsedTimeNs << "\t" << timeErrorPerElement << "\t" <<
      ellapsedTimeErrorNs << std::endl;

      std::free(vecX);
      std::free(vecY);
      std::free(vecRes);
}
```

Stop next to Start

Implementation + Compiler

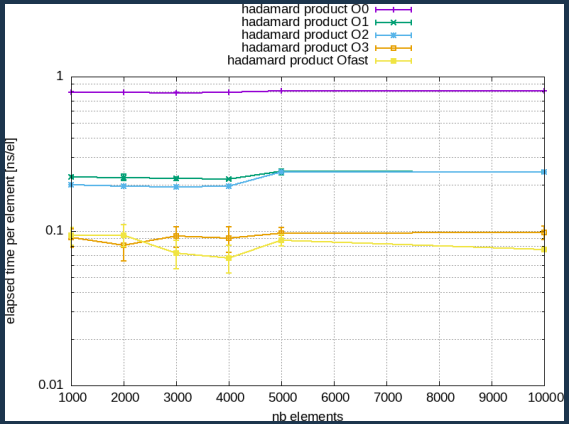
```
void evaluateHadamardProduct(size_t nbElement, size_t nbRepetition){
        //{On déclare nos tableaux et on les initialise :
        [illegible]
        [illegible]
        [illegible]

        [illegible]

        [illegible]
        [illegible]
        [illegible]

        size_t fullNbElement(nbElement*nbRepetition);
        //Stating the timer
        HiPeTime beginTime = phoenix_getTime();
        //Let's call the kernel several times
        [illegible]
                hadamard_product(vecRes, vecX, vecY, nbElement);
        }
        //Stop the timer
        NanoSecs elapsedTime(phoenix_getTime() - beginTime);
        double ellapsedTimeNs(elapsedTime.count()/((double)nbRepetition));
        double timePerElement = ellapsedTimeNs/((double)nbElement), timeErrorPerElement(0.0), ellapsedTimeErrorNs(0.0);
        std::cout << "Hadamard product naive ptr : nbElement = "<<fullNbElement<<", timePerElement = " << timePerElement << " ns/el ±
        "<<timeErrorPerElement<<", elapsedTime = " << ellapsedTimeNs << " ns ± "<<ellapsedTimeErrorNs << std::endl;
        std::cerr << nbElement << "\t" << timePerElement << "\t" << ellapsedTimeNs << "\t" << timeErrorPerElement << "\t" <<
        ellapsedTimeErrorNs << std::endl;

        [illegible]
        [illegible]
        [illegible]
}
```

Stop next to Start

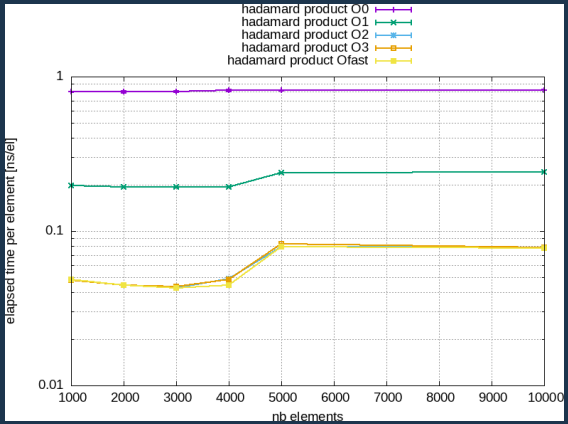
This happends because kernel is in the same file
Avoid problem if the result of the computation is used
No Problem with std::vector

The Hadamard product : Basic Options

G++ 11

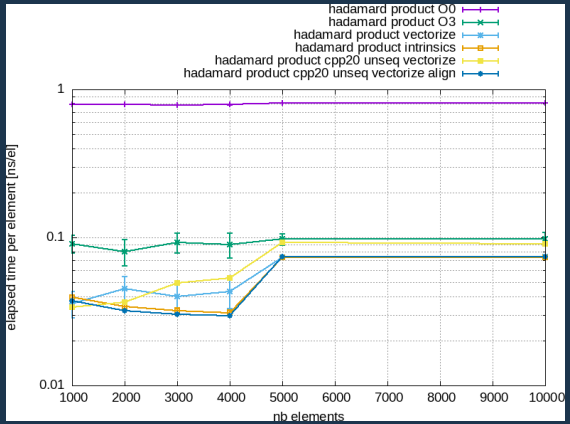


CLang++ 14

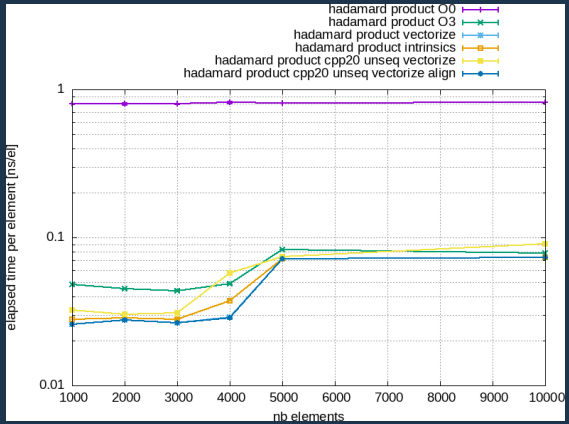


The Hadamard product : Vectorization

G++ 11



CLang++ 14



Performance Test Reminder

Performance Test Reminder

Are we measuring what we think we are measuring ?

Performance Test Reminder

Are we measuring what we think we are measuring ?

A performance test evaluates :

Performance Test Reminder

Are we measuring what we think we are measuring ?

A performance test evaluates :

With a given **method**

Performance Test Reminder

Are we measuring what we think we are measuring ?

A performance test evaluates :

With a given **method**

With a more or less **precise chrono**

Performance Test Reminder

Are we measuring what we think we are measuring ?

A performance test evaluates :

With a given **method**

The **implementation**

With a more or less **precise chrono**

Performance Test Reminder

Are we measuring what we think we are measuring ?

A performance test evaluates :

With a given **method**

The **implementation**

With a more or less **precise chrono**

Of more or less **quality**

Performance Test Reminder

Are we measuring what we think we are measuring ?

A performance test evaluates :

With a given **method**

The **implementation**

Of an **algorithm**

With a more or less **precise chrono**

Of more or less **quality**

Performance Test Reminder

Are we measuring what we think we are measuring ?

A performance test evaluates :

With a given **method**

The **implementation**

Of an **algorithm**

With a more or less **precise chrono**

Of more or less **quality**

More or less **relevant**

Performance Test Reminder

Are we measuring what we think we are measuring ?

A performance test evaluates :

- With a given **method**
- The **implementation**
- Of an **algorithm**
- **Compiled**
- With a more or less **precise chrono**
- Of more or less **quality**
- More or less **relevant**

Performance Test Reminder

Are we measuring what we think we are measuring ?

A performance test evaluates :

- With a given **method**
- The **implementation**
- Of an **algorithm**
- **Compiled**
- With a more or less **precise chrono**
- Of more or less **quality**
- More or less **relevant**
- With more or less **usefull flags**

Performance Test Reminder

Are we measuring what we think we are measuring ?

A performance test evaluates :

- With a given **method**
- The **implementation**
- Of an **algorithm**
- **Compiled**
- On a **computer**

- With a more or less **precise chrono**
- Of more or less **quality**
- More or less **relevant**
- With more or less **usefull flags**

Performance Test Reminder

Are we measuring what we think we are measuring ?

A performance test evaluates :

With a given **method**

The **implementation**

Of an **algorithm**

Compiled

On a **computer**

With a more or less **precise chrono**

Of more or less **quality**

More or less **relevant**

With more or less **usefull flags**

More or less **efficient**

Performance Test Reminder

Are we measuring what we think we are measuring ?

A performance test evaluates :

- With a given **method**
- The **implementation**
- Of an **algorithm**
- **Compiled**
- On a **computer**

- With a more or less **precise chrono**
- Of more or less **quality**
- More or less **relevant**
- With more or less **usefull flags**
- More or less **efficient**
- With more or less **shared ressources**

Performance Test Reminder

Are we measuring what we think we are measuring ?

A performance test evaluates :

- With a given **method**
- The **implementation**
- Of an **algorithm**
- **Compiled**
- On a **computer**
- With a **dataset**

- With a more or less **precise chrono**
- Of more or less **quality**
- More or less **relevant**
- With more or less **usefull flags**
- More or less **efficient**
- With more or less **shared ressources**

Performance Test Reminder

Are we measuring what we think we are measuring ?

A performance test evaluates :

- With a given **method**
- The **implementation**
- Of an **algorithm**
- **Compiled**
- On a **computer**
- With a **dataset**

- With a more or less **precise chrono**
- Of more or less **quality**
- More or less **relevant**
- With more or less **usefull flags**
- More or less **efficient**
- With more or less **shared ressources**
- More or less **representative** of the problem

Performance Test Reminder

Are we measuring what we think we are measuring ?

A performance test evaluates :

With a given **method**

The **implementation**

Of an **algorithm**

Compiled

On a **computer**

With a **dataset**

During an **ellapsed time**

With a more or less **precise chrono**

Of more or less **quality**

More or less **relevant**

With more or less **usefull flags**

More or less **efficient**

With more or less **shared ressources**

More or less **representative** of the problem

Performance Test Reminder

Are we measuring what we think we are measuring ?

A performance test evaluates :

With a given **method**

The **implementation**

Of an **algorithm**

Compiled

On a **computer**

With a **dataset**

During an **ellapsed time**

With a more or less **precise chrono**

Of more or less **quality**

More or less **relevant**

With more or less **usefull flags**

More or less **efficient**

With more or less **shared ressources**

More or less **representative** of the problem

Cleverly determined or not

Performance Test Reminder

Are we measuring what we think we are measuring ?

A performance test evaluates :

- With a given **method**
- The **implementation**
- Of an **algorithm**
- **Compiled**
- On a **computer**
- With a **dataset**
- During an **ellapsed time**
- On an **Operating System**

- With a more or less **precise chrono**
- Of more or less **quality**
- More or less **relevant**
- With more or less **usefull flags**
- More or less **efficient**
- With more or less **shared ressources**
- More or less **representative** of the problem
- **Cleverly** determined or not

Performance Test Reminder

Are we measuring what we think we are measuring ?

A performance test evaluates :

- With a given **method**
- The **implementation**
- Of an **algorithm**
- **Compiled**
- On a **computer**
- With a **dataset**
- During an **ellapsed time**
- On an **Operating System**

- With a more or less **precise chrono**
- Of more or less **quality**
- More or less **relevant**
- With more or less **usefull flags**
- More or less **efficient**
- With more or less **shared ressources**
- More or less **representative** of the problem
- **Cleverly** determined or not
- More or less **horrible** to use

Certification and compilers testing

Certification and compilers testing

Certification and compilers testing

Certification and compilers testing

Certification and compilers testing

Very difficult to develop — **Compiler** — **Certification Needed** — **Airplane Control**

Certification and compilers testing

Very difficult to develop → Compiler → Certification Needed → Airplane Control

CSmith

https://github.com/csmith-project/csmith

Certification and compilers testing

Very difficult to develop → Compiler → Certification Needed → Airplane Control

C Generator designed to **break compilers**

CSmith

https://github.com/csmith-project/csmith

Certification and compilers testing

Very difficult to develop → Compiler → Certification Needed → Airplane Control

C Generator designed to **break compilers** → CSmith

100+ bugs found in G++4 / Clang (2009-2011)

https://github.com/csmith-project/csmith

Certification and compilers testing

Very difficult to develop → Compiler → Certification Needed → Airplane Control

C Generator designed to **break compilers**

100+ bugs found in G++4 / Clang (2009-2011)

CSmith → **CompCert**

https://github.com/csmith-project/csmith

https://compcert.org/

Certification and compilers testing

Very difficult to develop → Compiler → Certification Needed → Airplane Control

C Generator designed to **break compilers**

CSmith

100+ bugs found in G++4 / Clang (2009-2011)

Mathematically Proved C Compiler

CompCert

https://github.com/csmith-project/csmith

https://compcert.org/

Conclusion

Conclusion

- Use **both** compilers
- Test **compilation options**
- Check with **Maqao** and **Valgrind**
- Check with **Performance Tests** (**MicroBenchmark**)

MAQAO