AUTOMATIC PARALLELISATION OF SOFTWARE USING GENETIC IMPROVEMENT

Bobby R. Bruce

INSPIRATION

Samsung Galaxy S7

INSPIRATION

Samsung Galaxy S7

Mali-T880 MP12

INSPIRATION

Samsung Galaxy S7

Mali-T880 MP12

Intel i7-2500K (overclocked to 5GHz)

BOBBY R. BRUCE

INSPIRATION

Samsung Galaxy S7

Mali-T880 MP12

Intel i7-2500K (overclocked to 5GHz)

INSPIRATION

Samsung Galaxy S7

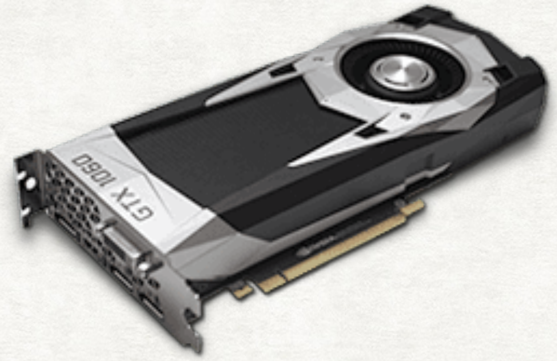
Mali-T880 MP12	Intel i7-2500K (overclocked to 5GHz)
	70 GFLOPs

INSPIRATION

Samsung Galaxy S7

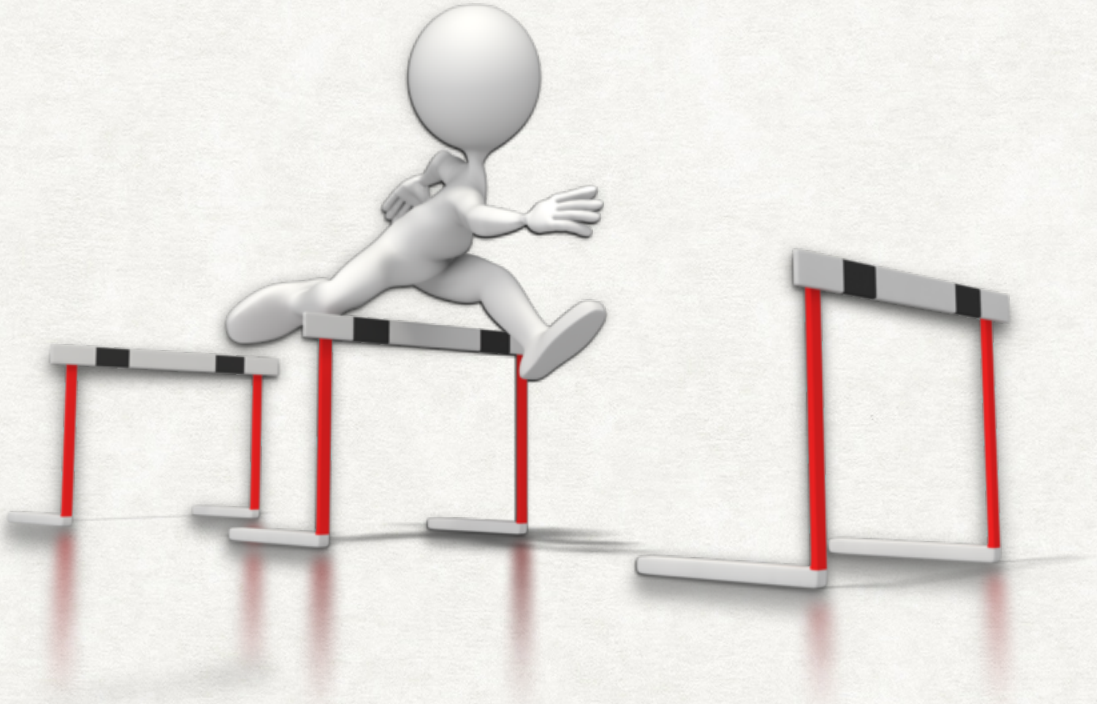
Mali-T880 MP12	Intel i7-2500K (overclocked to 5GHz)
265.2 GFLOPs	70 GFLOPs

INSPIRATION



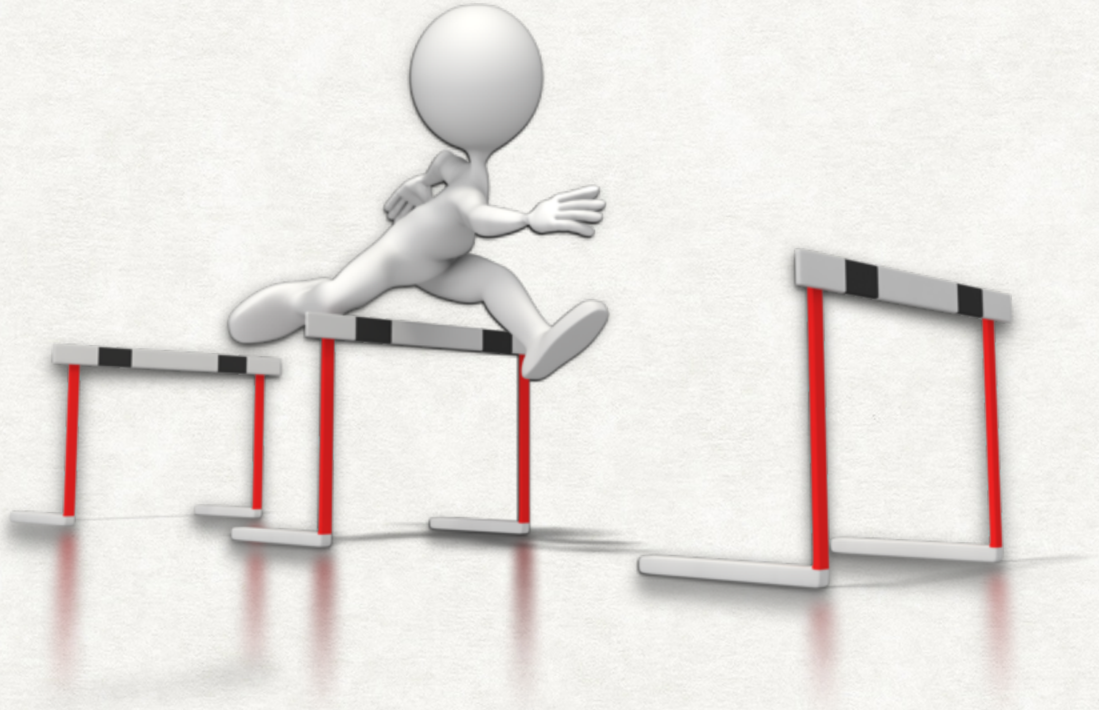
nVidia GTX 1060	Mali-T880 MP12	Intel i7-2500K (overclocked to 5GHz)
4327 GFLOPs	265.2 GFLOPs	70 GFLOPs

WHY DON'T WE UTILISE THIS POWERFUL HARDWARE?



- Developers lack the skills
- Hardware specialisation
- Developers' time is expensive; translating code to run on the GPU is expensive
- Getting decent optimisation requires manual trial and error

WHY DON'T WE UTILISE THIS POWERFUL HARDWARE?



An Automated approach would be ideal

- Developers lack the skills
- Hardware specialisation
- Developers' time is expensive; translating code to run on the GPU is expensive
- Getting decent optimisation requires manual trial and error

BACKGROUND: WHAT'S CURRENTLY AVAILABLE?

BACKGROUND: WHAT'S CURRENTLY AVAILABLE?

Domain	
Pros	
Cons	

BACKGROUND: WHAT'S CURRENTLY AVAILABLE?

Domain	Automatic Parallelisation Compilers
Pros	Does not require any skills, or knowledge of, parallelisation
Cons	Only targets very specific loops where dependencies are fully understood

BACKGROUND: WHAT'S CURRENTLY AVAILABLE?

Domain	Automatic Parallelisation Compilers	CUDA/OpenCL
Pros	Does not require any skills, or knowledge of, parallelisation	When implemented well offers the best performance
Cons	Only targets very specific loops where dependencies are fully understood	Difficult to learn, harder to master. Very Manual

BACKGROUND: WHAT’S CURRENTLY AVAILABLE?

Domain	Automatic Parallelisation Compilers	CUDA/OpenCL	Directive-based
Pros	Does not require any skills, or knowledge of, parallelisation	When implemented well offers the best performance	Considerably easier to implement.
Cons	Only targets very specific loops where dependencies are fully understood	Difficult to learn, harder to master. Very Manual	Still requires some skill, practise, and trial and error.

BACKGROUND: WHAT'S CURRENTLY AVAILABLE?

Domain	~~Automatic Parallelisation Compilers~~	CUDA/OpenCL	Directive-based
Pros	~~Does not require any skills, or knowledge of, parallelisation~~	When implemented well offers the best performance	Considerably easier to implement.
Cons	~~Only targets very specific loops where dependencies are fully understood~~	Difficult to learn, harder to master. Very Manual	Still requires some skill, practise, and trial and error.

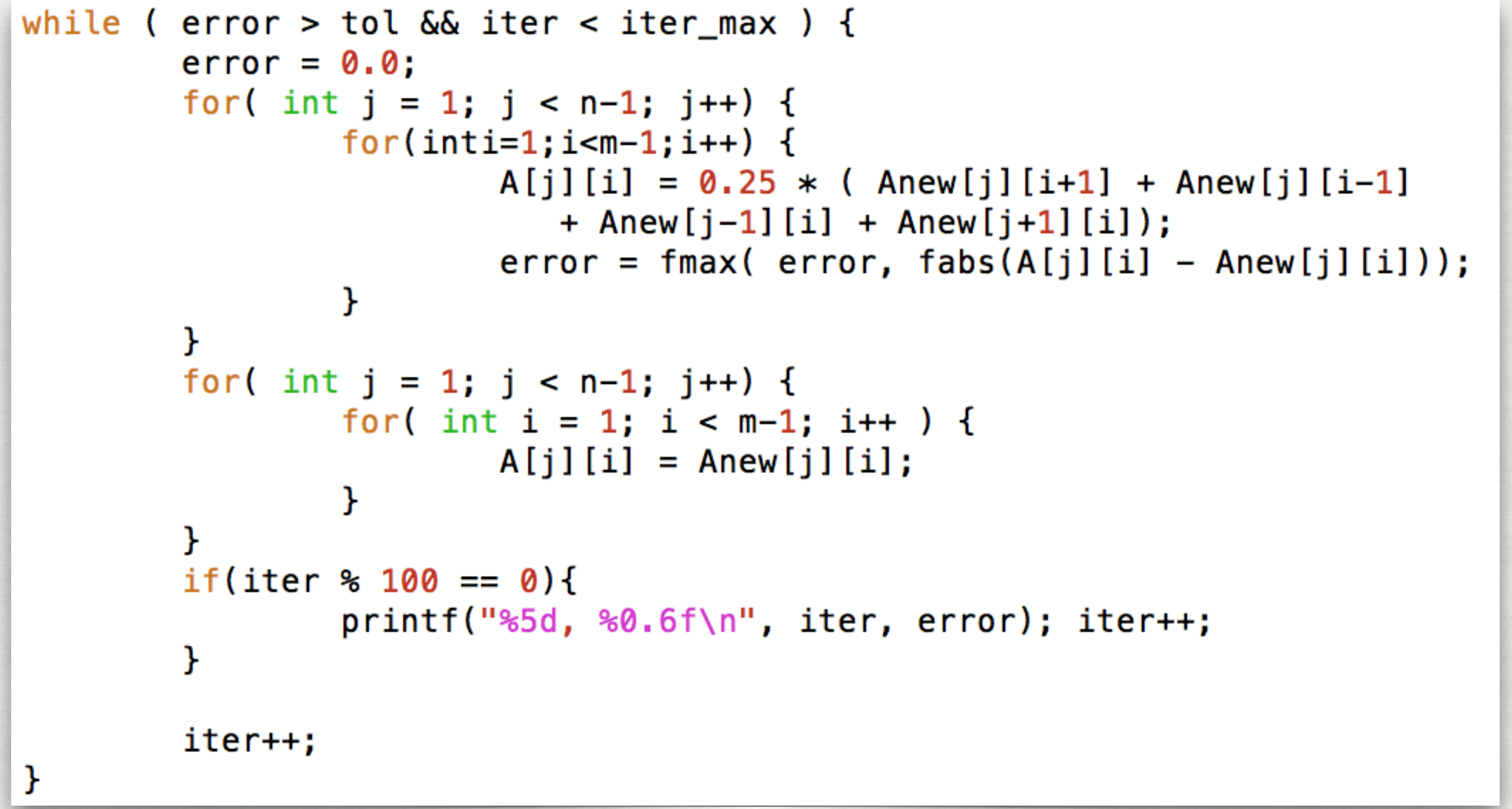
BACKGROUND: WHAT'S CURRENTLY AVAILABLE?

Domain	~~Automatic Parallelisation Compilers~~	~~CUDA/OpenCL~~	Directive-based
Pros	~~Does not require any skills, or knowledge of, parallelisation~~	~~When implemented well offers the best performance~~	Considerably easier to implement.
Cons	~~Only targets very specific loops where dependencies are fully understood~~	~~Difficult to learn, harder to master.~~ ~~Very Manual~~	Still requires some skill, practise, and trial and error.

BACKGROUND: WHAT'S CURRENTLY AVAILABLE?

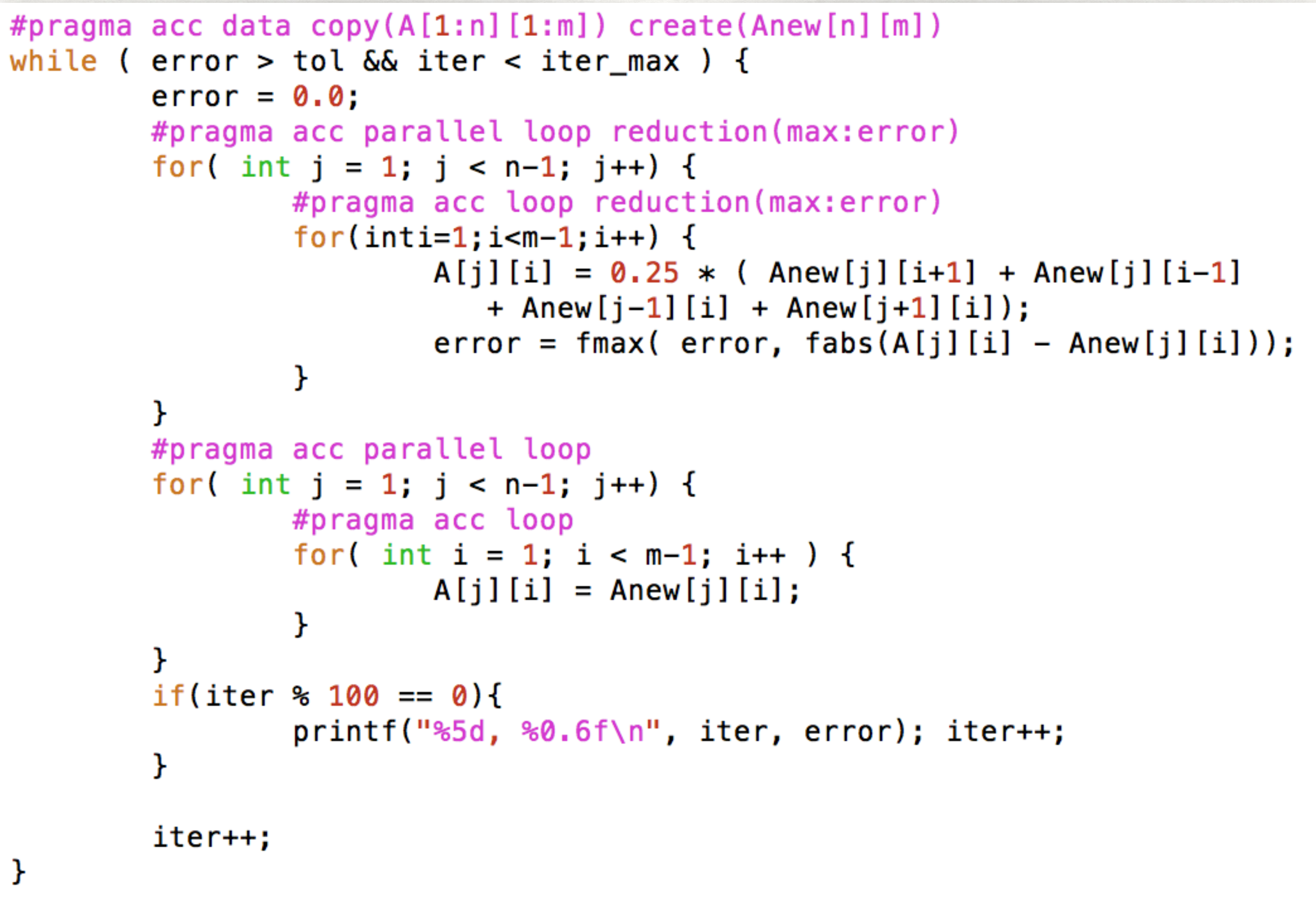
Domain	~~Automatic Parallelisation Compilers~~	~~CUDA/OpenCL~~	Directive-based
Pros	~~Does not require any skills, or knowledge of, parallelisation~~	~~When implemented well offers the best performance~~	Considerably easier to implement.
Cons	~~Only targets very specific loops where dependencies are fully understood~~	~~Difficult to learn, harder to master.~~ ~~Very Manual~~	Still requires some skill, practise, and trial and error.

BACKGROUND: OPENACC



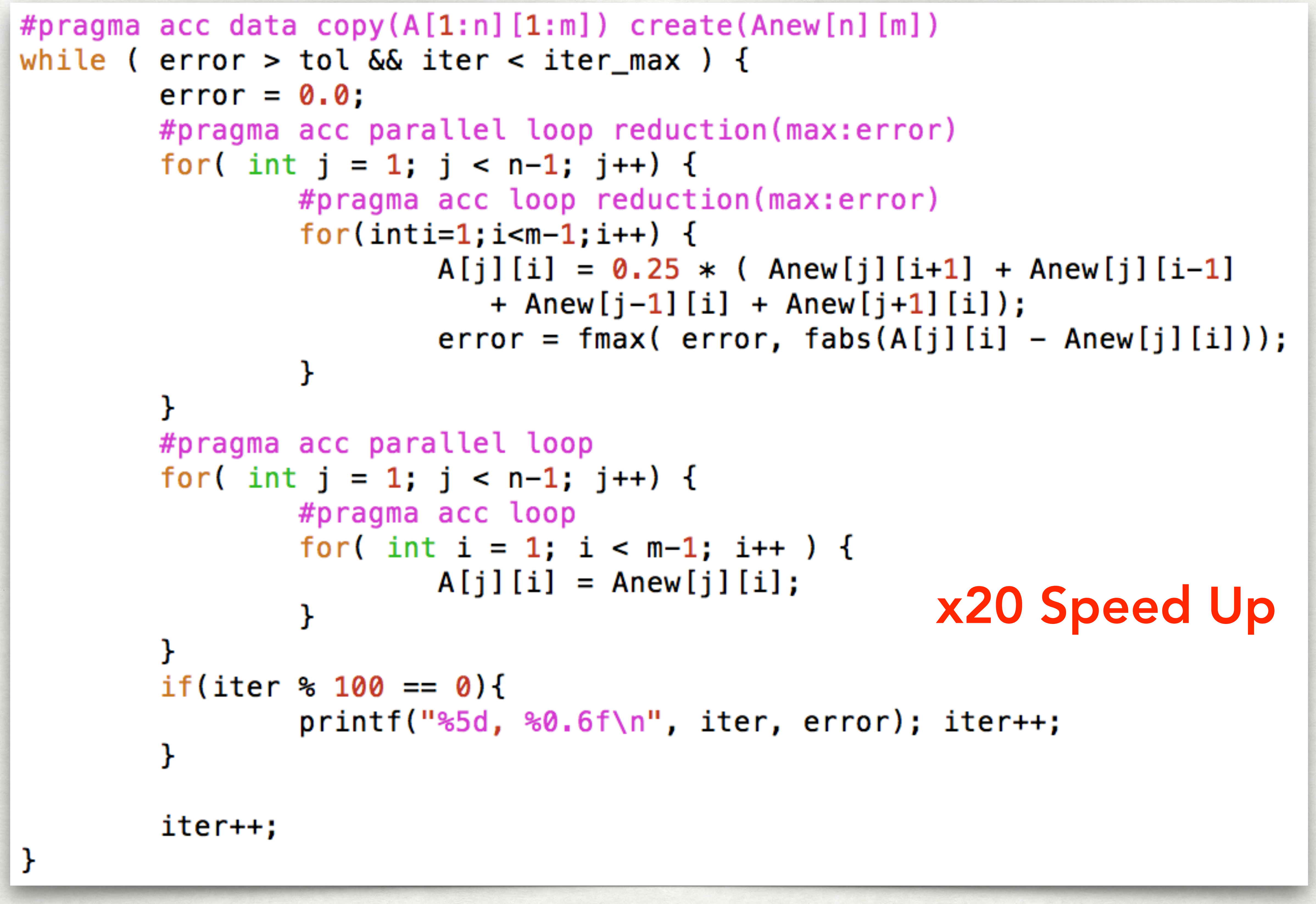
```
while ( error > tol && iter < iter_max ) {
        error = 0.0;
        for( int j = 1; j < n-1; j++) {
                for(inti=1;i<m-1;i++) {
                        A[j][i] = 0.25 * ( Anew[j][i+1] + Anew[j][i-1]
                           + Anew[j-1][i] + Anew[j+1][i]);
                        error = fmax( error, fabs(A[j][i] - Anew[j][i]));
                }
        }
        for( int j = 1; j < n-1; j++) {
                for( int i = 1; i < m-1; i++ ) {
                        A[j][i] = Anew[j][i];
                }
        }
        if(iter % 100 == 0){
                printf("%5d, %0.6f\n", iter, error); iter++;
        }

        iter++;
}
```

BACKGROUND: OPENACC



```
#pragma acc data copy(A[1:n][1:m]) create(Anew[n][m])
while ( error > tol && iter < iter_max ) {
        error = 0.0;
        #pragma acc parallel loop reduction(max:error)
        for( int j = 1; j < n-1; j++) {
                #pragma acc loop reduction(max:error)
                for(inti=1;i<m-1;i++) {
                        A[j][i] = 0.25 * ( Anew[j][i+1] + Anew[j][i-1]
                           + Anew[j-1][i] + Anew[j+1][i]);
                        error = fmax( error, fabs(A[j][i] - Anew[j][i]));
                }
        }
        #pragma acc parallel loop
        for( int j = 1; j < n-1; j++) {
                #pragma acc loop
                for( int i = 1; i < m-1; i++ ) {
                        A[j][i] = Anew[j][i];
                }
        }
        if(iter % 100 == 0){
                printf("%5d, %0.6f\n", iter, error); iter++;
        }

        iter++;
}
```

BACKGROUND: OPENACC



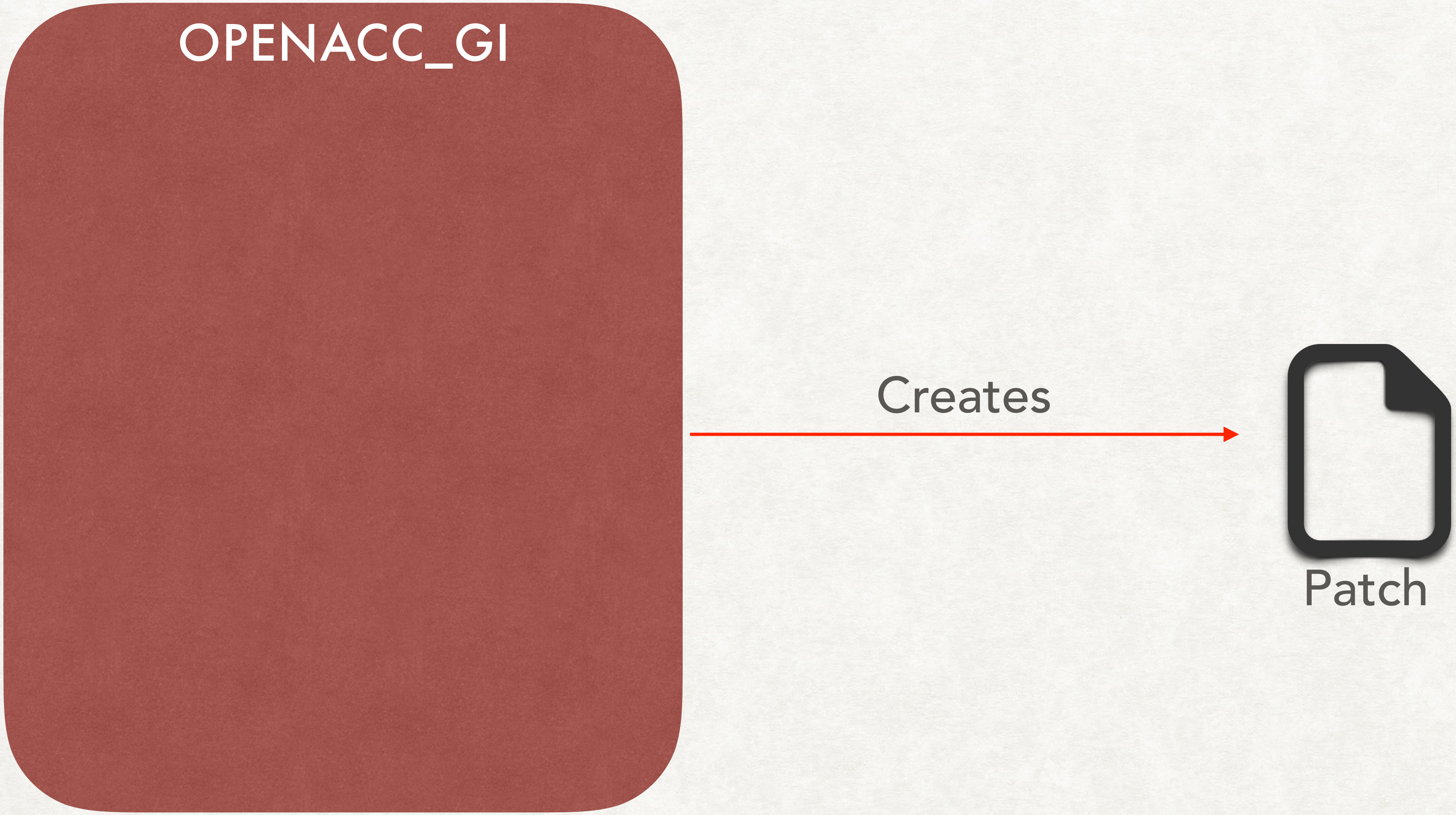
```
#pragma acc data copy(A[1:n][1:m]) create(Anew[n][m])
while ( error > tol && iter < iter_max ) {
        error = 0.0;
        #pragma acc parallel loop reduction(max:error)
        for( int j = 1; j < n-1; j++) {
                #pragma acc loop reduction(max:error)
                for(inti=1;i<m-1;i++) {
                        A[j][i] = 0.25 * ( Anew[j][i+1] + Anew[j][i-1]
                           + Anew[j-1][i] + Anew[j+1][i]);
                        error = fmax( error, fabs(A[j][i] - Anew[j][i]));
                }
        }
        #pragma acc parallel loop
        for( int j = 1; j < n-1; j++) {
                #pragma acc loop
                for( int i = 1; i < m-1; i++ ) {
                        A[j][i] = Anew[j][i];
                }
        }
        if(iter % 100 == 0){
                printf("%5d, %0.6f\n", iter, error); iter++;
        }

        iter++;
}
```

x20 Speed Up

OUR GOAL: AUTOMATICALLY ADD OPENACC DIRECTIVES

OPENACC_GI

OUR GOAL: AUTOMATICALLY ADD OPENACC DIRECTIVES

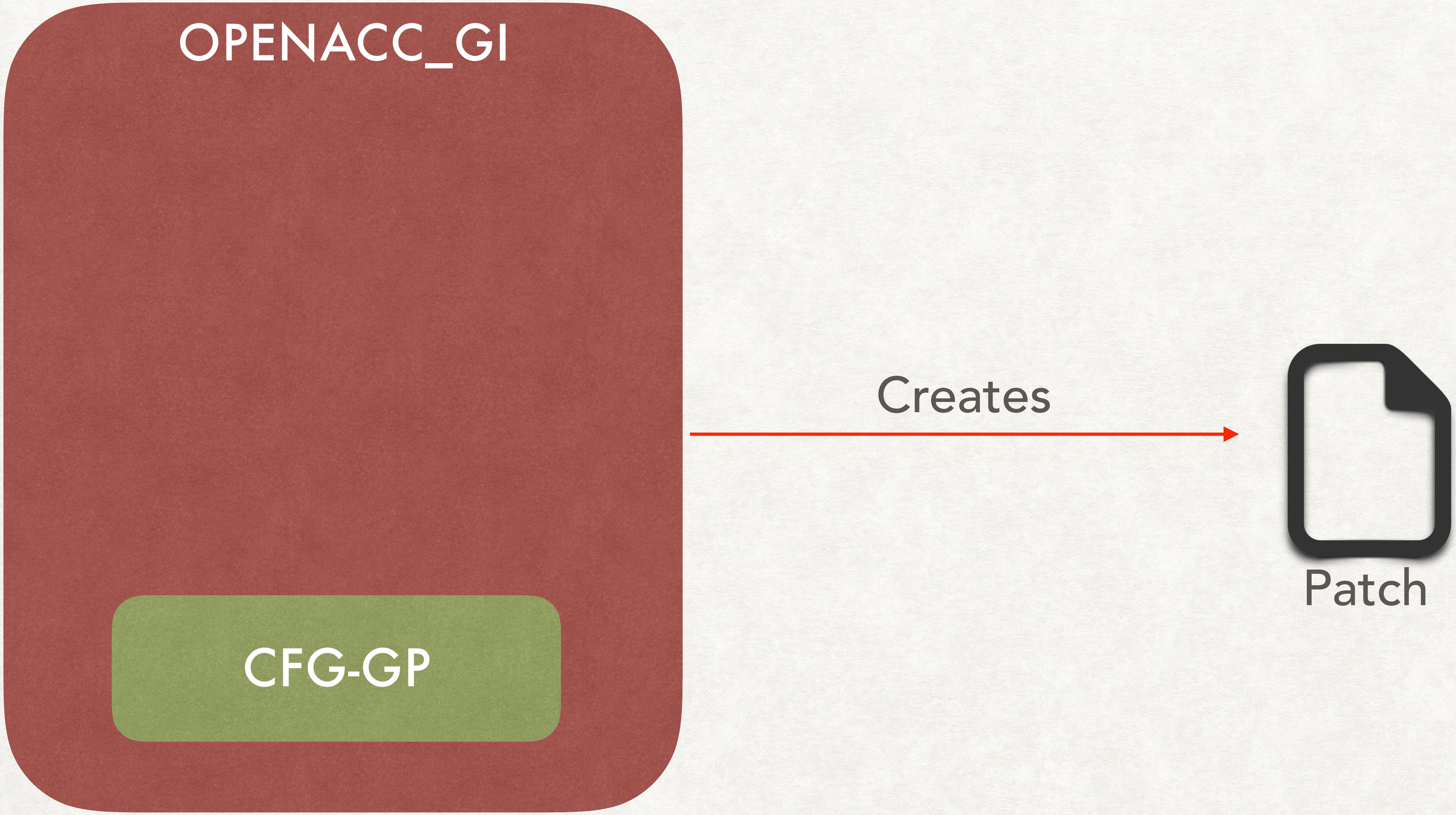


OPENACC_GI

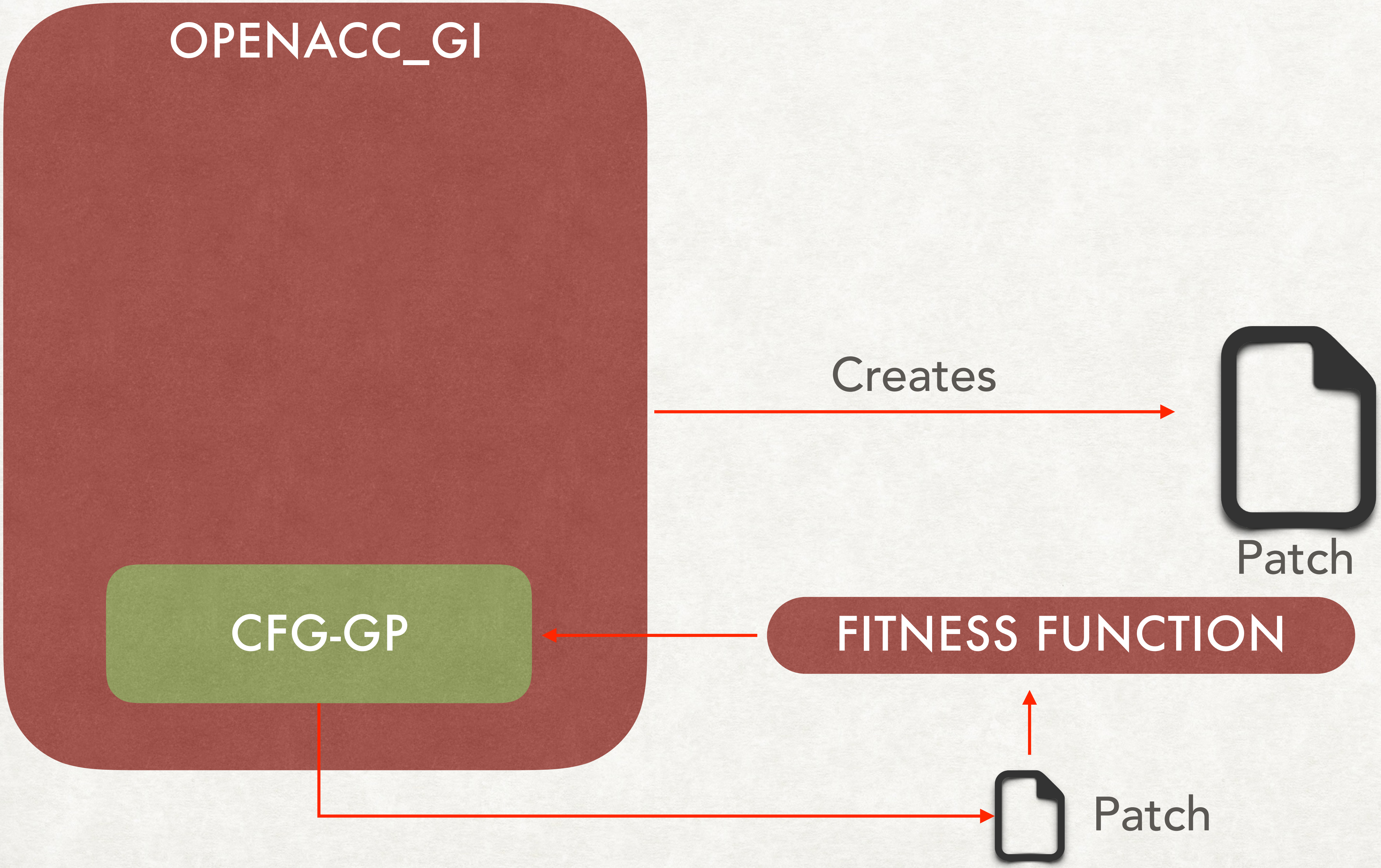
Creates

Patch

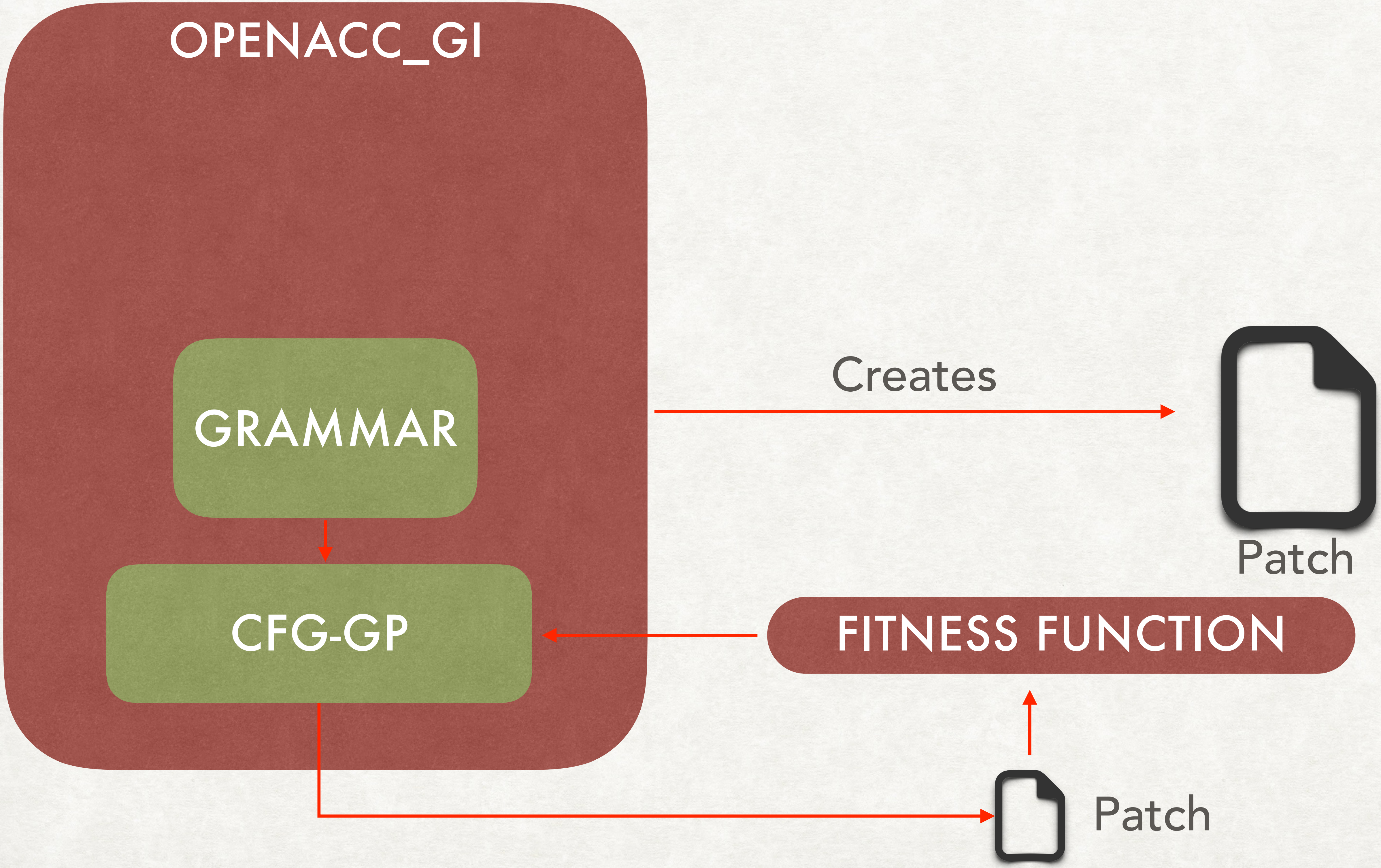
OUR GOAL: AUTOMATICALLY ADD OPENACC DIRECTIVES



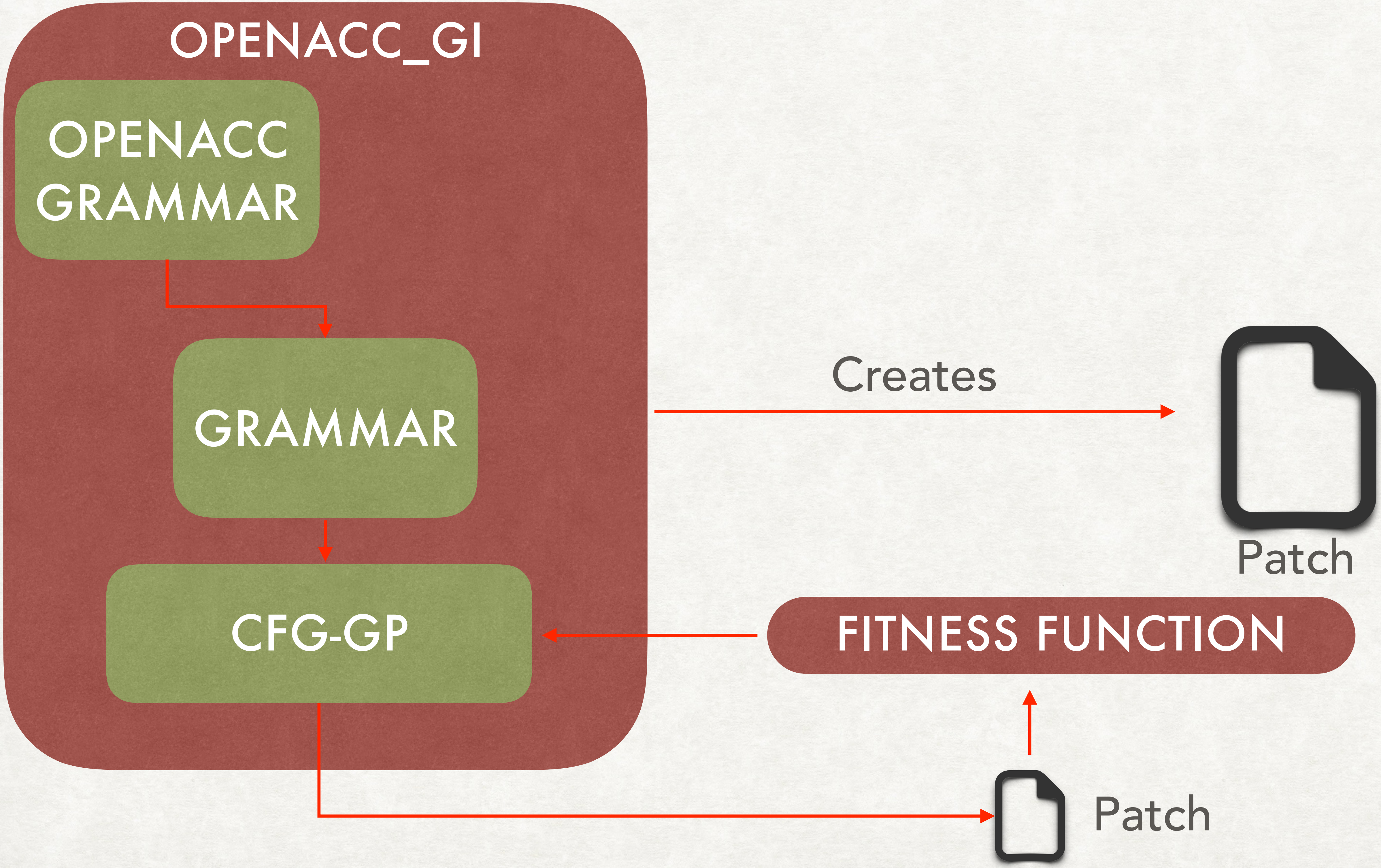
OUR GOAL: AUTOMATICALLY ADD OPENACC DIRECTIVES



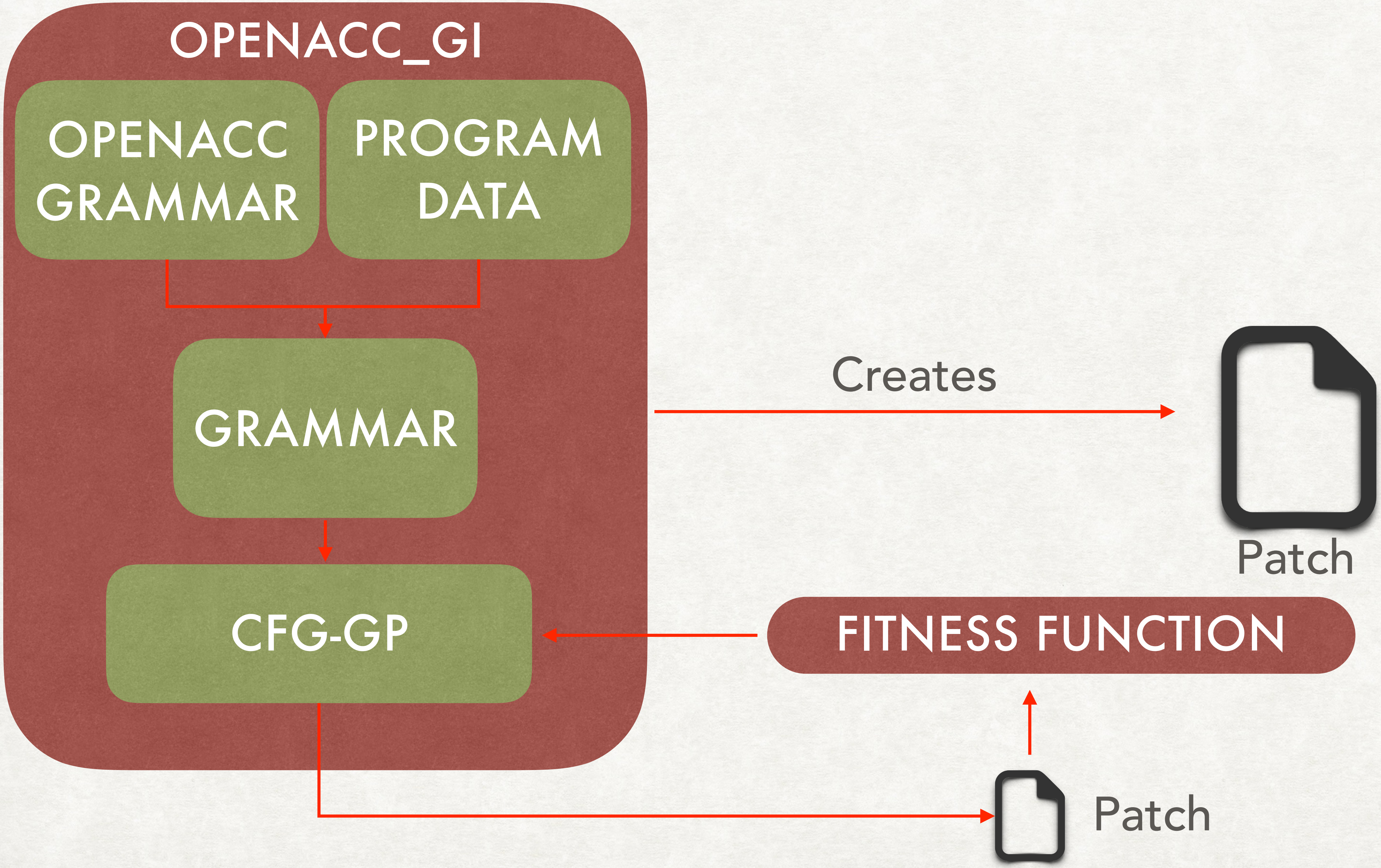
OUR GOAL: AUTOMATICALLY ADD OPENACC DIRECTIVES



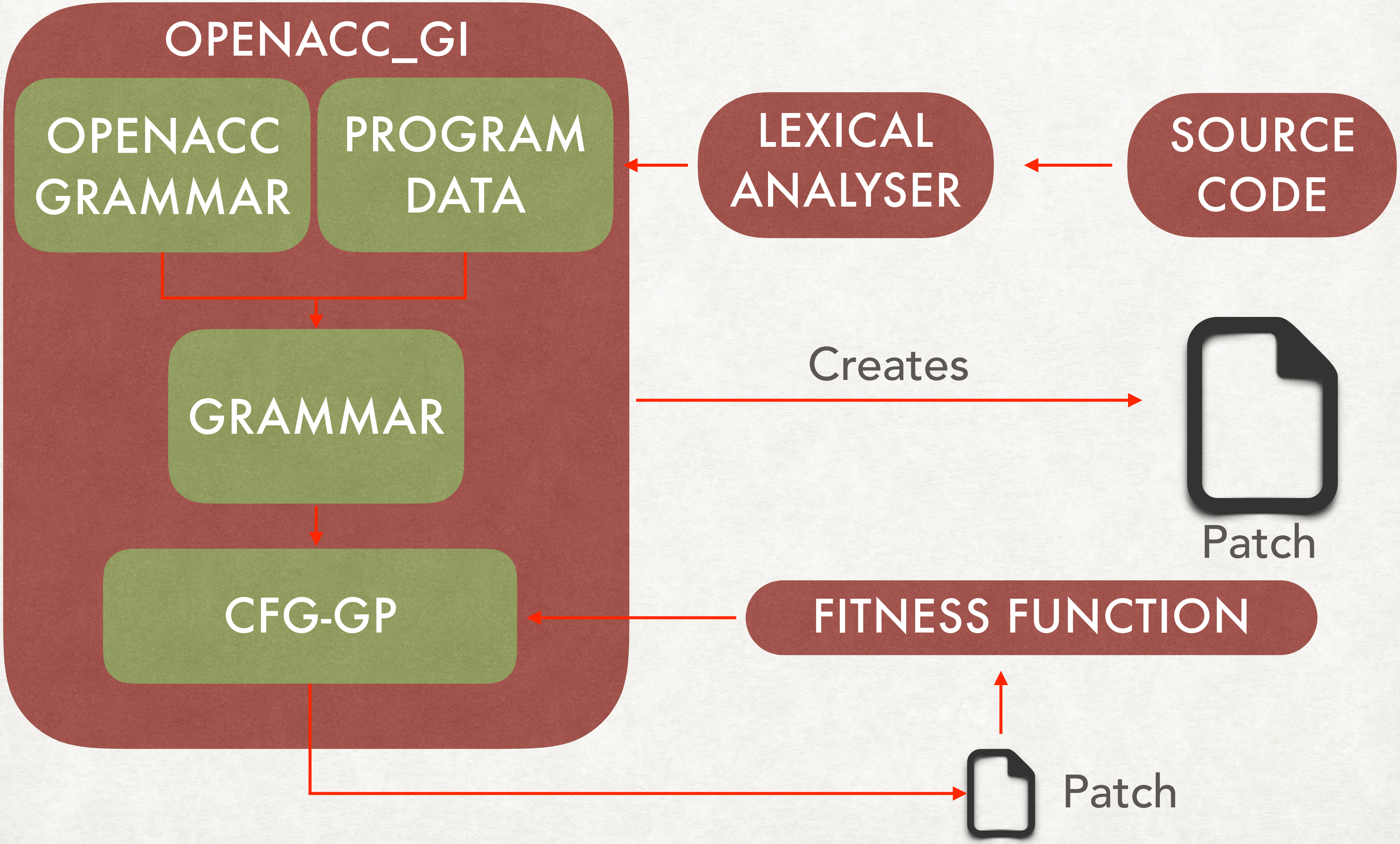
OUR GOAL: AUTOMATICALLY ADD OPENACC DIRECTIVES



OUR GOAL: AUTOMATICALLY ADD OPENACC DIRECTIVES



OUR GOAL: AUTOMATICALLY ADD OPENACC DIRECTIVES



GRAMMAR

```
<start> ::= <base> | <base> <start>
<base>     ::=  "#pragma acc " <choice>
<choice> ::=  "loop "<private> <loop_line_number>
<private> ::=  "private(" <variables> ") " | "  "
<variables> ::= <variable> | <variable> "," <variables>
<variable> ::= <variable_placeholder>
<variable_placeholder> ::= "1" | "2" | "3" | "4" | "5" | "6" | "7" …
```

GRAMMAR

```
<start> ::= <base> | <base> <start>
<base>    ::=  "#pragma acc " <choice>
<choice> ::=  "loop "<private> <loop_line_number>
<private> ::=  "private(" <variables> ") " | "  "
<variables> ::= <variable> | <variable> "," <variables>
<variable> ::= <variable_placeholder>
<variable_placeholder> ::=  "1" | "2" | "3" | "4" | "5" | "6" | "7"  …
<loop_line_number> ::=  "15@example1.c" | "145@example2.c"
```

GRAMMAR

```
<start> ::= <base> | <base> <start>
<base>    ::=  "#pragma acc " <choice>
<choice> ::=  "loop "<private> <loop_line_number>
<private> ::=  "private(" <variables> ") " | "  "
<variables> ::= <variable> | <variable> "," <variables>
<variable> ::= <variable_placeholder>
<variable_placeholder> ::=  "1" | "2" | "3" | "4" | "5" | "6" | "7" …
<loop_line_number> ::=  "15@example1.c" | "145@example2.c"
```

#pragma acc loop private(1,2) 15@example1.c

GRAMMAR

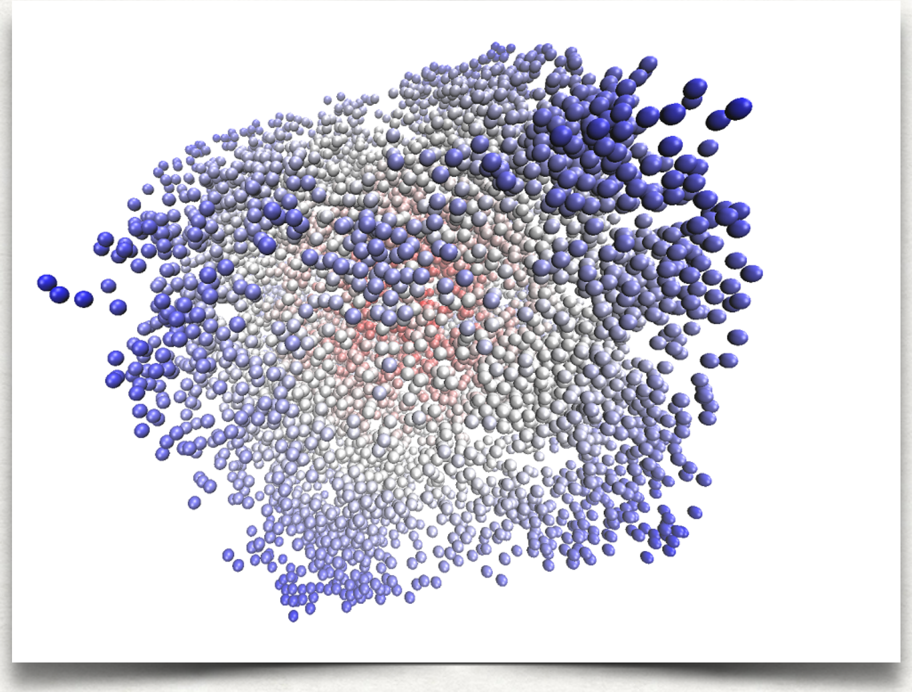
```
<start> ::= <base> | <base> <start>
<base>    ::=  "#pragma acc " <choice>
<choice> ::=  "loop "<private> <loop_line_number>
<private> ::=  "private(" <variables>  ") " | "  "
<variables> ::= <variable> | <variable>  "," <variables>
<variable> ::= <variable_placeholder>
<variable_placeholder> ::=  "1" | "2" | "3" | "4" | "5" | "6" | "7"  …
<loop_line_number> ::=  "15@example1.c" | "145@example2.c"
```

GRAMMAR

```
<start> ::= <base> | <base> <start>
<base>    ::=  "#pragma acc " <choice>
<choice> ::=  "loop "<private> <loop_line_number>
<private> ::=  "private(" <variables> ") " | "  "
<variables> ::= <variable> | <variable> "," <variables>
<variable> ::= <variable_placeholder>
<variable_placeholder> ::= "1" | "2" | "3" | "4" | "5" | "6" | "7" …
<loop_line_number> ::= "15@example1.c" | "145@example2.c"
```

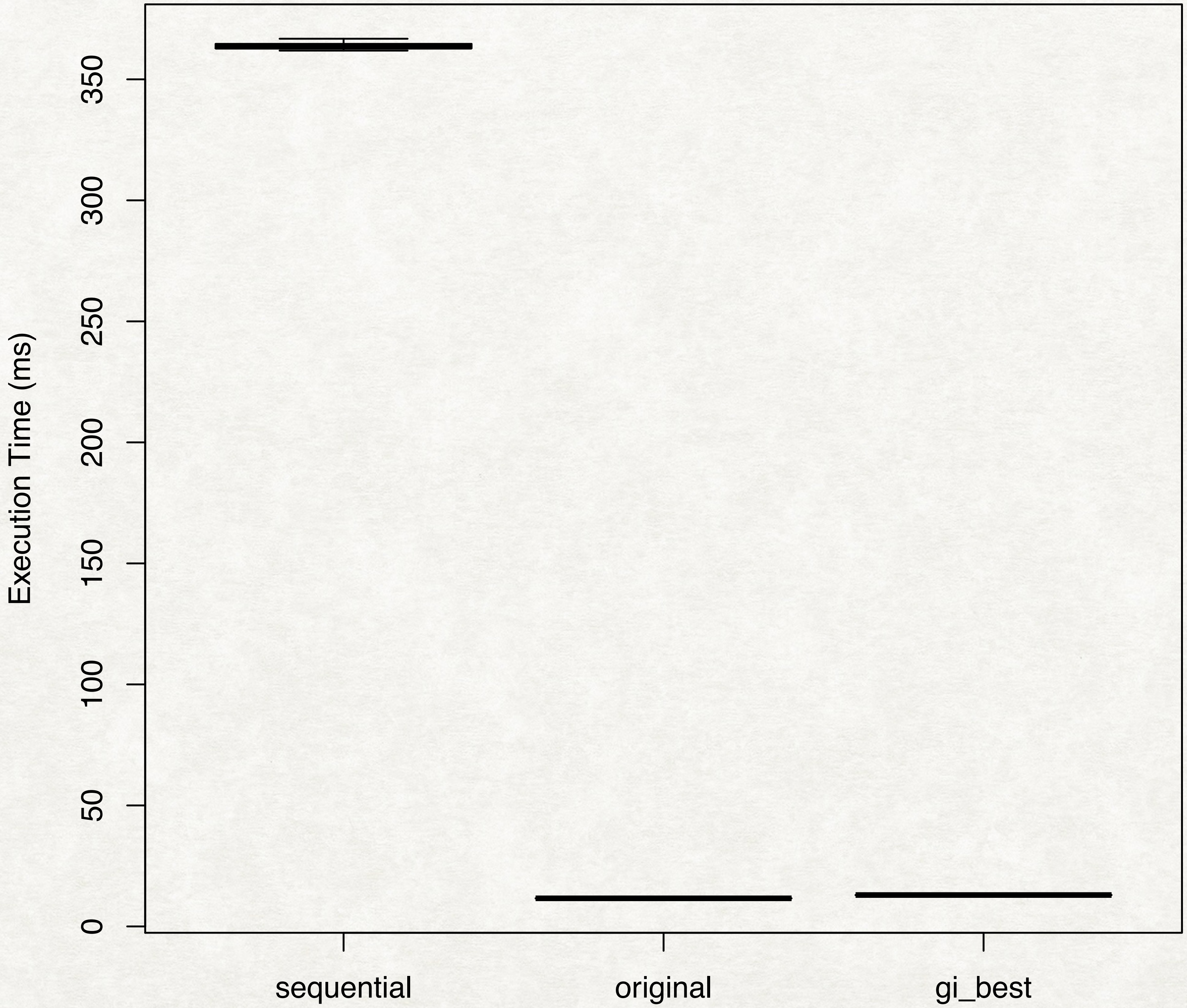
```
—- example1.c
+++ example1.c
@@ -15,0 +15,1 @@
+ #pragma acc loop private(x,y)
```

INITIAL INVESTIGATION

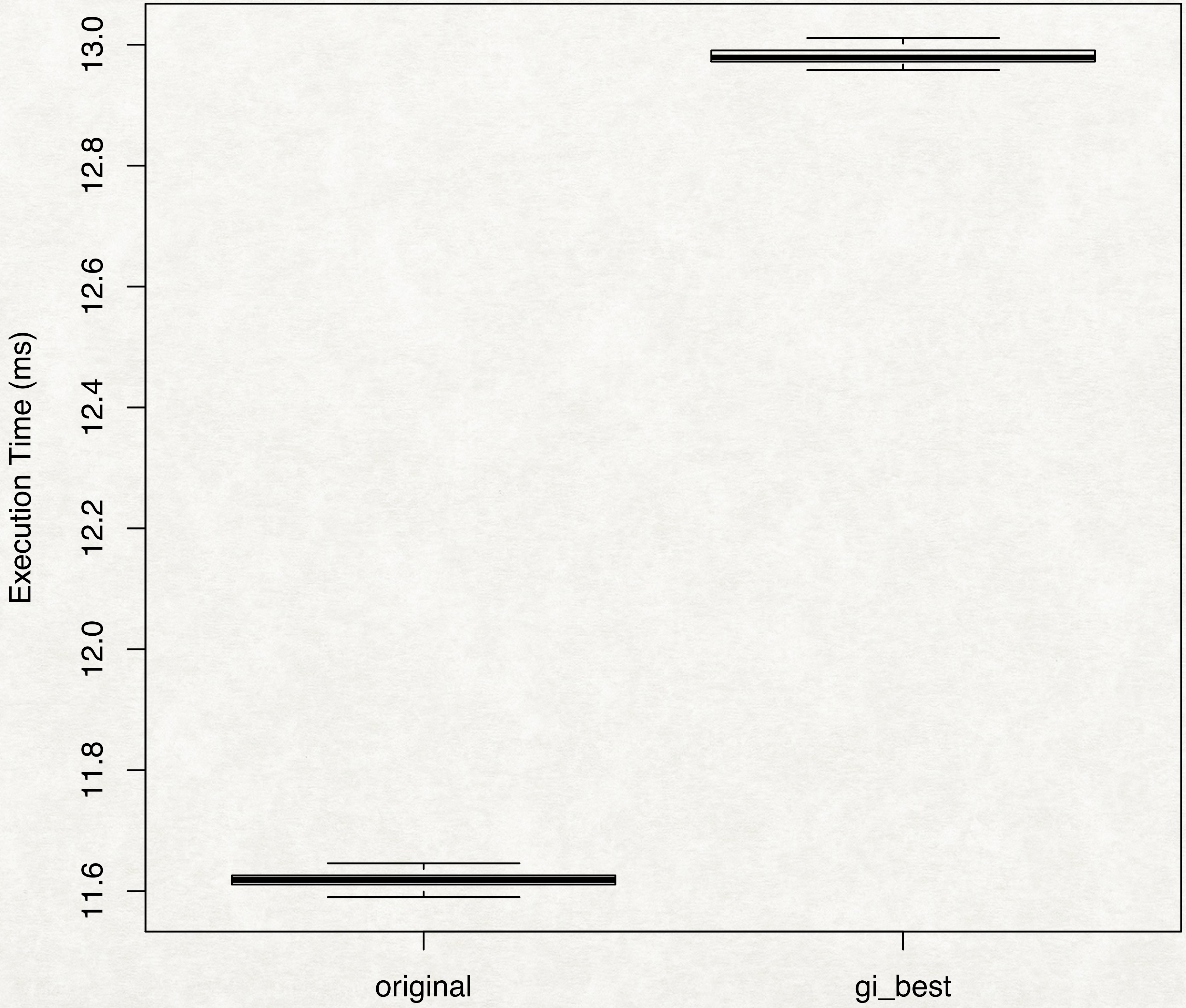


- Chose to run a very small example as a sanity check
- nVidia provide an n-body simulation example already containing OpenACC directives
- These directives were stripped for openacc to replicate
- Ran for 100 generations with population of 100

RESULTS

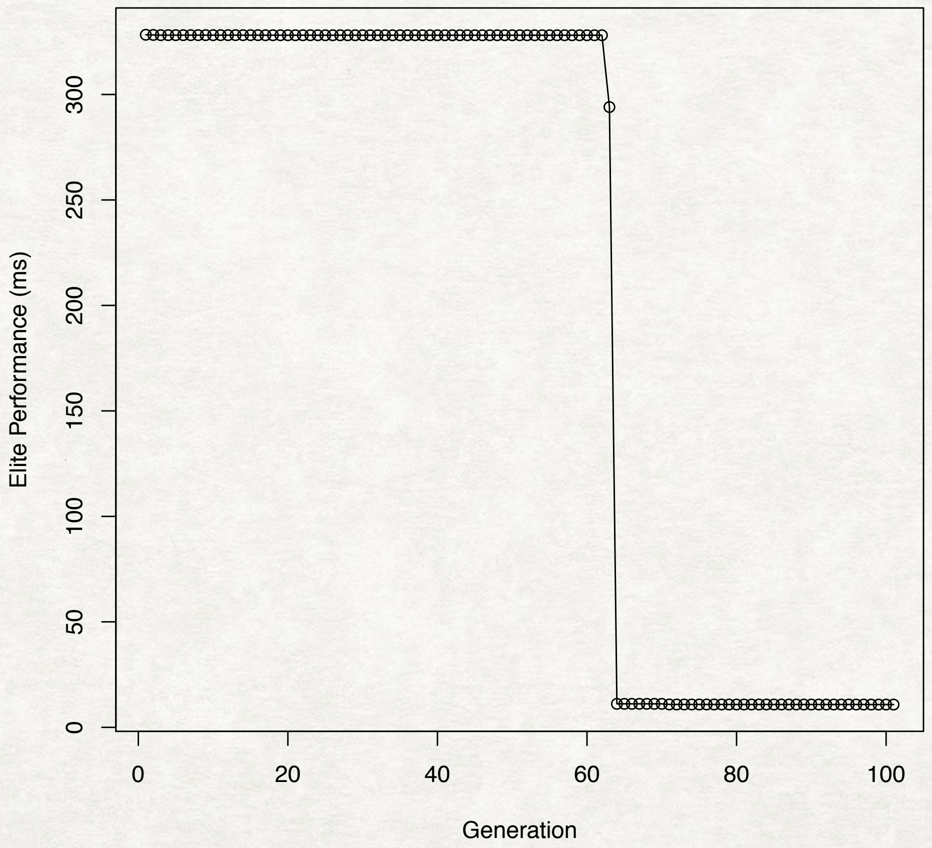

Execution Time (ms)
0
50
100
150
200
250
300
350
sequential
original
gi_best

RESULTS


Execution Time (ms)
13.0
12.8
12.6
12.4
12.2
12.0
11.8
11.6
original
gi_best

RESULTS: OTHER NOTES

- Seems like much of the gain is due to random search
- We'd like to be able to beat human-written alternatives
- This example is very small, future investigations will show how well the tool scales



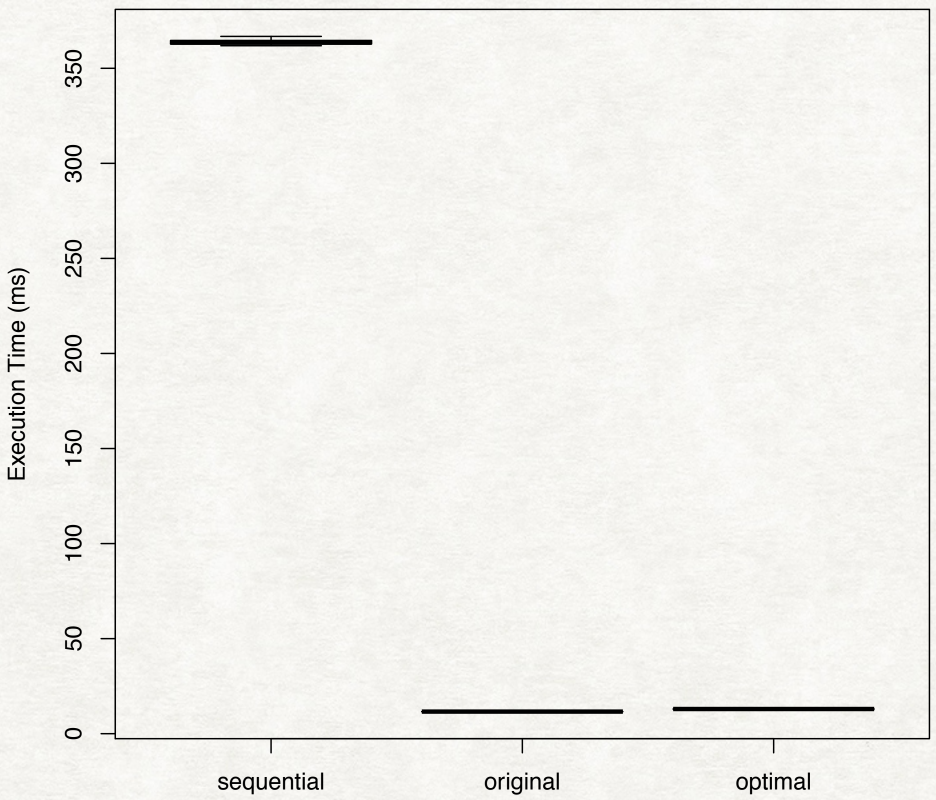
CURRENT/FUTURE WORK

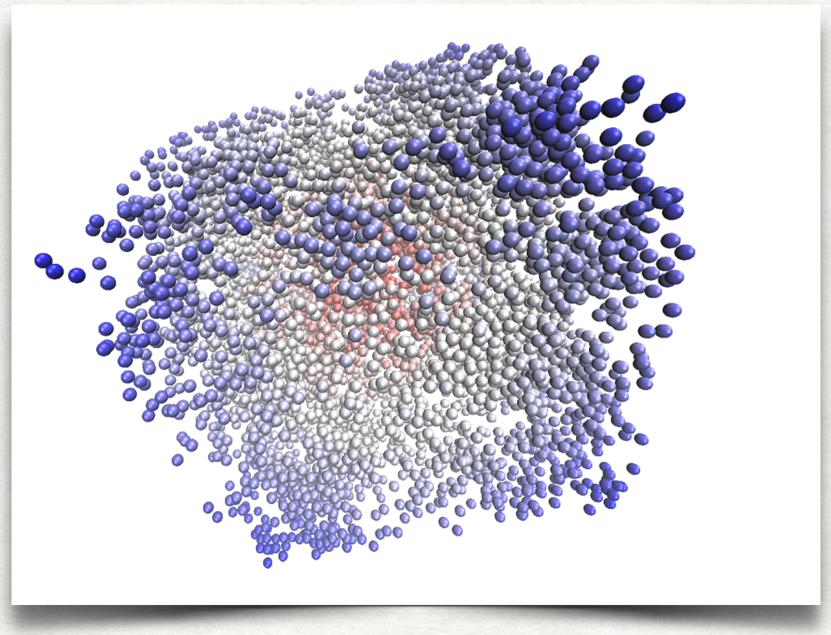
- Currently applying the tool to larger
- At present can only work with C/C++, expanding code to work with FORTRAN

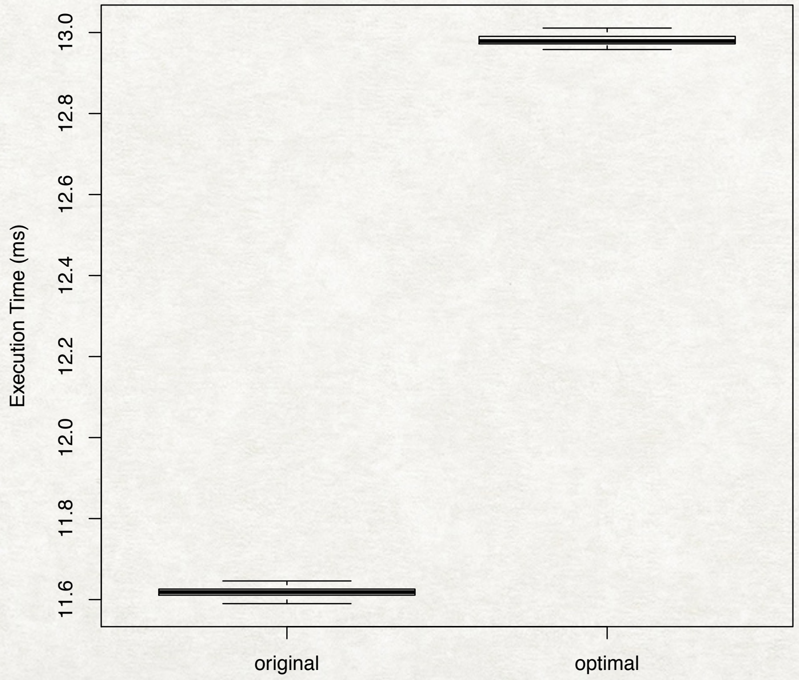
Possible Improvements:

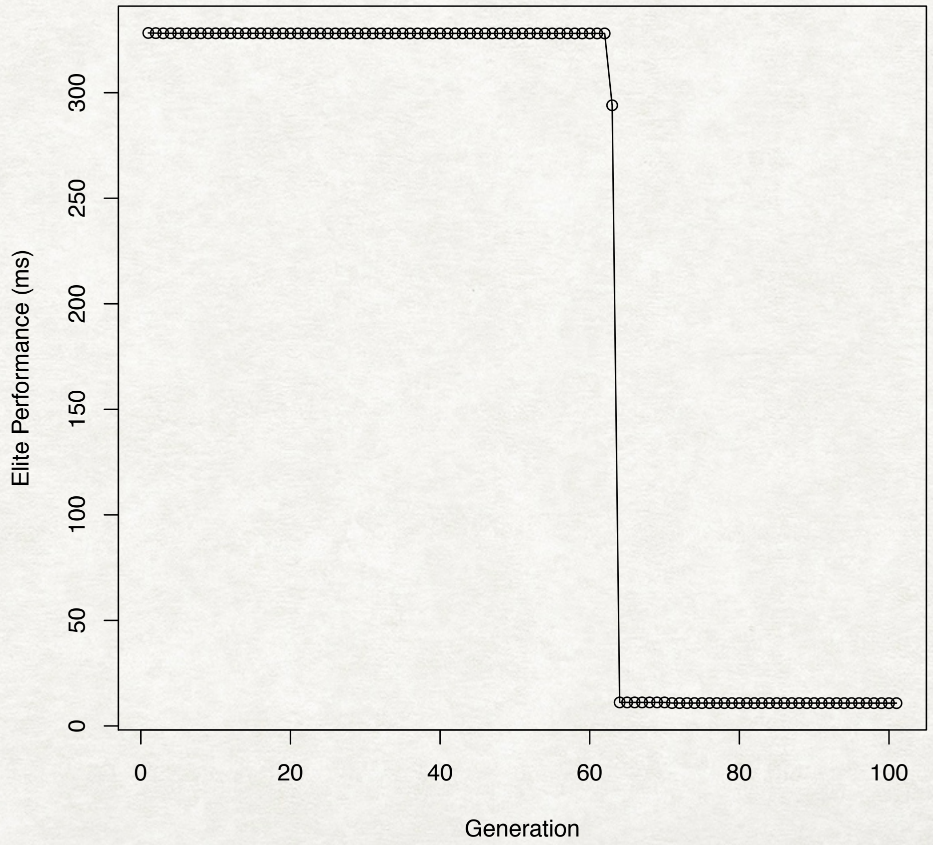
- Seed initial generation with basic solutions
- Introduce some clever profiling
- Get working with OpenMP as well as OpenACC

ANY QUESTIONS?









BOBBY R. BRUCE