Low-hanging fruit

Carlos Bederián, carlos.bederian@unc.edu.ar

Nicolás Wolovick, nicolasw@famaf.unc.edu.ar

CCAD Centro de Computación de Alto Desempeño

UNC Universidad Nacional de Córdoba

Motivación

- En HPC se puede llegar relativamente lejos con poco trabajo
  - Buen uso de las herramientas disponibles
  - Paralelización con directivas
  - Bibliotecas
- ¡Y se puede llegar a ninguna parte después de mucho trabajo!
  - "Optimizar" **todo** a mano

Agenda

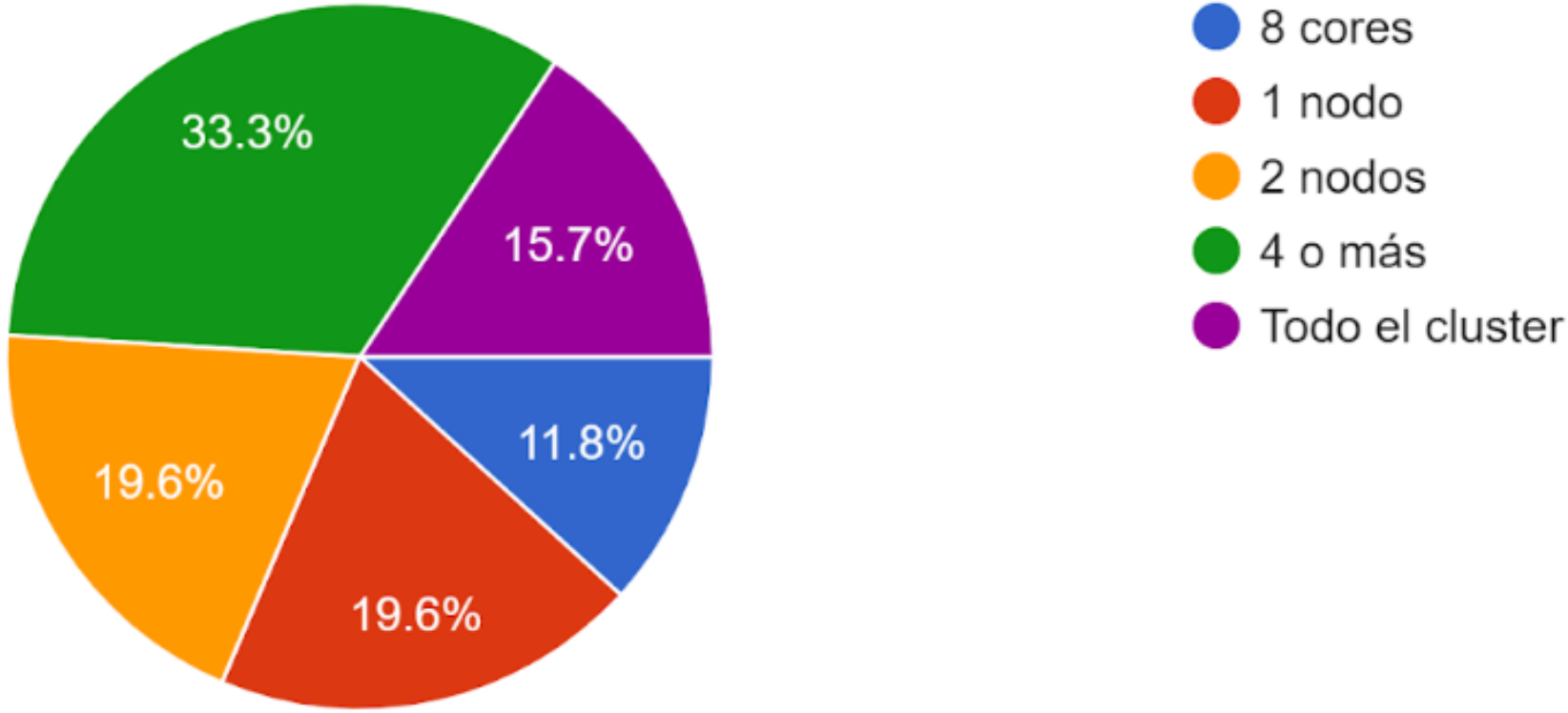
- Hoy: Mejorar performance fácilmente con el compilador
  - Y todos los rabbit holes donde no meterse
- Mañana: Herramientas para ver por dónde seguir

Un segundo...

¿Esto no era de HPC?

Encuesta a usuarios CCAD

¿Cuántos recursos podría aprovechar si contara con uso exclusivo por 1 mes?



Compiladores

Un compilador moderno es:

- Un front end que convierte el código en un lenguaje particular a una representación intermedia (IR)
  - Preprocesador
  - Análisis léxico
  - Análisis sintáctico
  - Análisis semántico
- Un middle end que transforma la representación intermedia
  - Análisis y optimización
- Un back end que transforma la representación intermedia a la arquitectura en cuestión
  - Optimización para la arquitectura
  - Generación de código

GCC

Clásica suite de compiladores de GNU

- Iniciado en 1987 por Richard Stallman
- Licencia GPL (copyleft)
- Ada, C, C++, D, Fortran, Go, Objective-C
- Soporta prácticamente cualquier arquitectura
- Última versión: GCC 9.1.0 (abril 2019)

LLVM

Colección modular de tecnología para compiladores

- Iniciado en 2003 por Chris Lattner
- Licencia UIUC/NCSA
- Altamente modular
  - Reemplazo de back end de GCC
  - Clang (Apple), compilador completo para C, C++, Objective-C
  - CUDA, ISPC, Julia, OpenCL, Rust, Swift...
  - Flang/f18 (NVIDIA), compilador Fortran 2018
- Soporte de arquitecturas generalmente provisto por los fabricantes mismos
  - Aunque no siempre en código abierto
- Generalmente compatible con GCC en parámetros y extensiones
- Última versión: LLVM 8.0 (marzo 2019)

Intel

Suite propietaria de compiladores para HPC

- Propietario
- *Caro* (para nosotros)
  - Licencias especiales para estudiantes, proyectos open source, educadores
- C, C++, Fortran
- Optimizado para arquitecturas Intel
  - ...y a veces ofuscado para AMD
- Última versión: Parallel Studio XE 2019 Update 4 (junio 2019)

Consiguiendo compiladores

Cada versión nueva de un compilador tiene:

- Bugfixes
- Soporte de versiones nuevas de lenguajes en el front end
- Más o mejores optimizaciones

Formas de obtener la última versión:

- Esperar una nueva versión de la distribución de linux
- Usar repositorios third-party
- Bajarlo y compilarlo
  - Spack, Easybuild

Optimizaciones del compilador

- Problema NP-completo o indecidible
- Transformaciones de código por otro equivalente a distintas escalas
  - Peephole
  - Local (bloque básico)
  - Loop
  - Global (función)
  - Interprocedural

Strength reduction, instruction combining

Reemplazo de expresiones por otras equivalentes con menor costo computacional

```
4 * a                          a << 2
n % 2 == 0                     n & 1 == 0
i++; i++                       i += 2
x = 0                          x = x^x
for (i=0; i<N; ++i) a[i]=0;    memset(a, 0, N*sizeof(a[0]))
```

Punto flotante

El compilador generalmente no intenta hacer nada con operaciones de punto flotante.

- D.Goldberg, What Every Computer Scientist Should Know About Floating-Point Arithmetic

`-ffast-math` al...rescate?

- Reordenamiento y reemplazo de instrucciones
- Permite aproximaciones con inversos
- Actualización de `errno` en funciones de punto flotante
- Asume que no hay ceros con signo
- Apaga checks de `NaN` o cero en algunos lugares
- Ignora excepciones del hardware
- Denormals truncados a cero (en algunos compiladores)

En los compiladores Intel, `fp-model fast` viene prendido por defecto.

Adivina adivinador

¿Cuánto demora en ejecutar este código?

```c
#define N 1073741824ull

int main() {
    double *a = malloc(N * sizeof(double));
    double sum = 0.0;
    for (size_t i = 0; i < N; ++i) {
        sum += a[i];
    }
    return 0;
}
```

Dead code elimination

El compilador elimina código que no se ejecuta o que **no afecta el resultado**

- Análisis agresivo
- Activado en `-O1`
  - Pesadilla para la gente que piensa en seguridad
- ¡No avisa cuando lo hace!

Constant folding

El compilador calcula en tiempo de compilación los resultados que puede.

Constant propagation

El compilador lleva rastro de los valores de las variables.

- La combinación con otras optimizaciones es poderosa:
  - Dead code elimination: Saltea guardas innecesarias
  - Strength reduction: Convierte operaciones caras como %
  - Constant folding: Pre-calcula expresiones más complejas
- Fija cotas de loops

Constant folding (...)

```
int constant_sum(void) {
    int sum = 0;
    for (int i = 1; i <= 5; ++i) {
        sum += i;
    }
    return sum;
}
```

Live range analysis

Constant propagation recargado. El compilador lleva rastro de rangos de valores de las variables.

```
int square(int x) {
    int y = x * x;
    if (y < 0) {
        printf("Acá no se llega nunca");
    }
    return y;
}
```

Common subexpression elimination

El compilador ahorra trabajo sacando factor común.

- Aplica al cálculo de posiciones en matrices y sus vecinos

```
float cse(float a, float b, float c) {
    float x = (a * b) - c;
    float y = (a * b) + c;
    return x / y;
}
```

Variable renaming

El compilador genera copias de variables que se reutilizan con propósitos independientes.

- Permite paralelismo en la ejecución
- Optimización que viene gratis con la conversión a forma SSA

Invariant hoisting

El compilador detecta código invariante dentro de un loop y lo mueve afuera.

Induction variable analysis

El compilador analiza cómo se utiliza la variable de inducción dentro del loop y transforma las expresiones

```
void zero_odd(float a[]) {
    for (int i = 0; i < N; ++i) {
        a[2*i + 1] = 0.0f;
    }
}
```

Loop unrolling

El compilador despliega un loop de manera que cada iteración opere sobre múltiples elementos.

- Mucho paralelismo disponible
- Altísimo costo en cache de instrucciones

```
void array_sum(float a[], float b[], float c[]) {
    for (int i = 0; i < N; ++i) {
        c[i] = a[i] + b[i];
    }
}

void array_sum_unrolled(float a[], float b[], float c[]) {
    int i;
    for (i = 0; i < N - i%4; i += 4) {
        c[i+0] = a[i+0] + b[i+0];
        c[i+1] = a[i+1] + b[i+1];
        c[i+2] = a[i+2] + b[i+2];
        c[i+3] = a[i+3] + b[i+3];
    }
    for (; i < N; ++i) {
        c[i] = a[i] + b[i];
    }
}
```

Loop unswitching

De haber una condición invariante dentro de un loop, el compilador la extrae y generar dos versiones del loop.

```
void array_divide(float a[], float d) {
    for (int i = 0; i < N; ++i) {
        if (d == 0.0f) {
            a[i] = 0.0f;
        } else {
            a[i] = a[i] / d;
        }
    }
}
```

Loop peeling

Separar iteraciones con comportamiento distinto (generalmente la primera)

- En este momento, `gcc`, `clang` e `icc` no saben separar las iteraciones del borde del resto.

```
void stencil(float a[], float b[], int N) {
    for (unsigned int i=0; i<N; ++i) {
        if (i==0) {
            b[i] = a[i+1]/2;
        } else if (i==N-1) {
            b[i] = a[i-1]/2;
        } else { // 0<i<N-1
            b[i] = (a[i-1]+a[i+1])/2;
        }
    }
}
```

Loop fission, loop fusion

Separar o unir loops independientes que corren sobre el mismo rango.

```
void init(float a[], float b[], float c[]) {
    for (int i=0; i<N; ++i) {
        a[i] = f();
        b[i] = g();
        c[i] = h();
    }
}
```

Matrices en memoria

Las matrices se tienen que guardar en memoria, que es unidimensional.

Para una matriz `A` NxN:

- Fortran: `A(y,x)` → `x*N + y` ("column-major")
- C: `A[y][x]` → `y*N + x` ("row-major")

Esto es importante porque la memoria y el procesador operan sobre segmentos contíguos de **64 bytes** $[i * 64, (i + 1) * 64)$.

- 8 doubles, 16 floats
- ¡Lo que no se utiliza es ancho de banda de memoria malgastado!

Loop interchange

Intercambio de loops anidados para mejor localidad.

- Ojo acá: Saber las razones y escribirlo bien desde un principio no afecta la legibilidad
- No siempre le acierta el compilador (GCC en particular)

```
float a[N][N], b[N][N], c[N][N];

void matmul() {
    for (int y=0; y<N; ++y)
        for (int x=0; x<N; ++x)
            for (int k=0; k<N; ++k)
                c[y][x] += a[y][k] * b[k][x];
}
```

Loop blocking/tiling

Particionar las iteraciones para obtener mejor localidad de memoria.

Ejemplo patológico: Transponer una matriz

- ¡11% de cache misses!

```
float A[N][N], At[N][N];

void transpose() {
    for (int y=0; y<N; ++y)
        for (int x=0; x<N; ++x)
            At[x][y] = A[y][x];
}
```

Con loop blocking

```
float A[N][N], At[N][N];

void transpose() {
    for (by=0; by<N; by+=BY) {
        for (bx=0; bx<N; bx+=BX) {
            for (y=by; y<by+BY; ++y) {
                for (x=bx; x<bx+BX; ++x) {
                    At[x][y] = A[y][x];
                }
            }
        }
    }
}
```

Inlining

Reemplazar un llamado a función por directamente copiar el cuerpo de la función dentro del código del llamador

- Se ahorra todo el proceso de llamado a función
  - Pasaje de parámetros, prólogo, epílogo
- Se paga con duplicación de código
  - Presión sobre el cache de instrucciones
- El compilador estima con heurísticas si conviene
- Nota: Si la función es visible fuera del módulo, también se genera la versión estándar
- **No tiene sentido usar macros para esto**

Link-time optimization

Las optimizaciones generalmente se limitan a **una unidad de compilación** por cómo se llama al compilador.

Con LTO (GCC: `-flto`, Intel: `-ipo`) sólo se corre el front end sobre cada unidad de compilación, y el resto de las fases se dejan para el momento de link de todo el programa.

- Nota: Requiere toolchain moderna

Autovectorización

Los procesadores tienen instrucciones vectoriales que operan sobre conjuntos de elementos de longitud fija.

- En un procesador con AVX-512, no utilizar instrucciones vectoriales para doubles es tirar ~80% de la performance
- Los elementos generalmente tienen que estar *contíguos en memoria*
- Los elementos tienen que ser independientes

Código autovectorizado

1. (A veces) Loop inicial escalar hasta llegar a dirección alineada
2. Loop vectorizado, procesa múltiples elementos por ciclo
3. Loop escalar para el resto de los elementos

Nota: Si la cantidad de elementos es pequeña, esto es más lento

¿Y si no autovectoriza?

El autovectorizador no funciona para código relativamente complejo: **acá es donde hay que optimizar**

1. Revisar mensajes de diagnóstico del compilador por los que no vectorizó un loop y arreglarlos
   - Motivo #1: las cosas están mal dispuestas en memoria
2. Indicarle al compilador con directivas OpenMP SIMD
3. Usar otro lenguaje más amigable para vectorizar (e.g. SYCL, ISPC)
4. Vectorizar a mano con intrinsics

Autoparalelización

El compilador analiza si las iteraciones de un loop son independientes, y las reparte entre hilos si conviene.

- En general no funciona, pero nunca está de más probar...

Selección de arquitectura

El compilador tiene un modelo de las unidades de ejecución del procesador:

- Costo y latencia de cada instrucción
- Puertos de ejecución
- Sets de instrucciones soportados

Si uno no le dice nada, el compilador genera código para *cualquier* procesador

- En X86-64, esto es un procesador con SSE y SSE2...de 2003.

Feedback-driven optimization

Muchas de las optimizaciones se deciden según heurísticas. FDO se trata de observar el programa en funcionamiento para compilarlo con más conocimiento.

1. Primera pasada: Compilar el programa con instrumentación del compilador para obtener métricas
   - Alternativa nueva para GCC/Clang: AutoFDO, obtiene métricas de una compilación normal del programa utilizando `perf`.
2. Correr el programa tratando de ejercitar todo el código
3. Segunda pasada: Compilar nuevamente el programa pasándole al compilador las métricas obtenidas

Juntando todo

- Queremos aplicar todas las optimizaciones posibles (`-O3`)
- Las de punto flotante también (GCC: `-ffast-math`, Intel: `-fp-model fast=2 -no-prec-div`)
- Optimizar entre distintas funciones y módulos (GCC: `-flto`, Intel: `-ipo`)
- Para el procesador que tenemos (GCC: `-march=native`, Intel: `-xHost`)
- Obteniendo métricas para optimizar mejor (GCC: `-fprofile-generate`, Intel: `-prof-gen`)
  - ...o usando la información que obtuvimos (GCC: `-fprofile-use`, Intel: `-prof-use`)
- Incluir información de debugging para el profiler (`-g`)
- Ver si loop blocking ayuda (GCC: `-floop-block`)
- Avisame dónde no pudiste vectorizar (GCC: `-fopt-info-vec-missed`, Intel: `-qopt-report -qopt-report-phase=vec`)

Python

- CPython es un intérprete *lento*
  - Se llaman funciones y bibliotecas implementadas en Fortran, C o C++ para cualquier cosa pesada
- NumPy para todo
  - ¿A qué BLAS y LAPACK llama?
  - ¿Con qué compilador se compiló el código Fortran?
- Si no alcanza, probar con Numba

Distribuciones de Python

- La que vino con la distro de Linux
  - Usa el BLAS instalado (ATLAS u OpenBLAS)
  - Compilado con gfortran de la distribución
- Anaconda
  - Mantenida por Continuum
  - Mezcla de package manager con virtualenv
  - Opción de MKL u OpenBLAS
  - En general el Python más rápido
- A mano

Numba

Permite marcar código Python+Numpy con decoradores para que sea compilado con LLVM.

- Aplicable sólo para un subconjunto del lenguaje
- Soporta correr en GPUs y paralelizar

```
import numba
import numpy

@numba.jit
def sum(x):
    total = 0
    for i in range(x.shape[0]):
        total += x[i]
    return total

x = numpy.arange(10_000_000);
%time sum(x)
%time sum(x)
```

Julia

Lenguaje interpretado específicamente creado para aplicaciones científicas.

- Sintaxis familiar para usuarios de Fortran
  - 1-based arrays
- Herramientas modernas
  - Jupyter notebooks
- Ecosistema creciente
- Diseñado para performance
  - El código en realidad se compila con LLVM

Aplicaciones HPC

1. Bajar binarios optimizados por los desarrolladores
    - Si es necesario recompilar (e.g. por usar otro MPI), ver si tienen opciones de compilador sugeridas
2. Buscar otra gente del rubro que lo haya hecho
    - XCONFIGURE (de Intel, para Intel)
    - HPC Advisory Council Best Practices
    - Spack
    - Easybuild
3. Soborne a su sysadmin favorito

El elefante en la habitación

¿Y TensorFlow?

Q&A