# Automatic Partitioning of Sequential Applications Driven by Domain-Independent Kernels

*José M. Andión*, Manuel Arenaz, and Juan Touriño

Computer Architecture Group
Department of Electronics and Systems
University of A Coruña
Spain

# Index

- Introduction: Motivation & Foundations
- Domain-Independent Kernel-based IR
  - Kernel-based Data Dependence Graph (KDDG)
  - Kernel-based Control Flow Graph (KCFG)
- Automatic Partitioning
- Case Studies
  - Sobel Edge Filter
  - EQUAKE from SPEC CPU2000
- Conclusions & Future Work

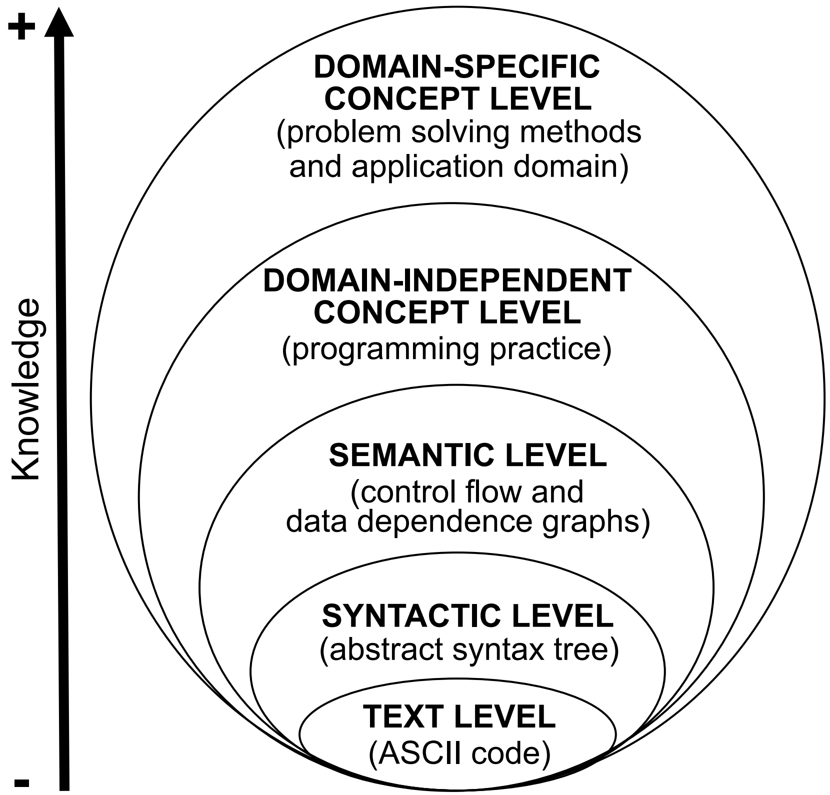
# Index

- **Introduction: Motivation & Foundations**
- Domain-Independent Kernel-based IR
  - Kernel-based Data Dependence Graph (KDDG)
  - Kernel-based Control Flow Graph (KCFG)
- Automatic Partitioning
- Case Studies
  - Sobel Edge Filter
  - EQUAKE from SPEC CPU2000
- Conclusions & Future Work

# Motivation

- Emergence and widespread use of multicore and manycore systems
- Partitioning of sequential applications
  - By the developer
  - Domain-specific languages
  - Complex and time-consuming
  - In-depth knowledge about application and computer architecture
- We propose
  - New kernel-based IR
  - Automatic approach to application partitioning

# Domain-Independent Computational Kernels




Reuse of platform-optimized parallel software libraries
Software re-engineering
Program synthesis

Induction variable substitution
Parallelizing transformations for inductions, reductions

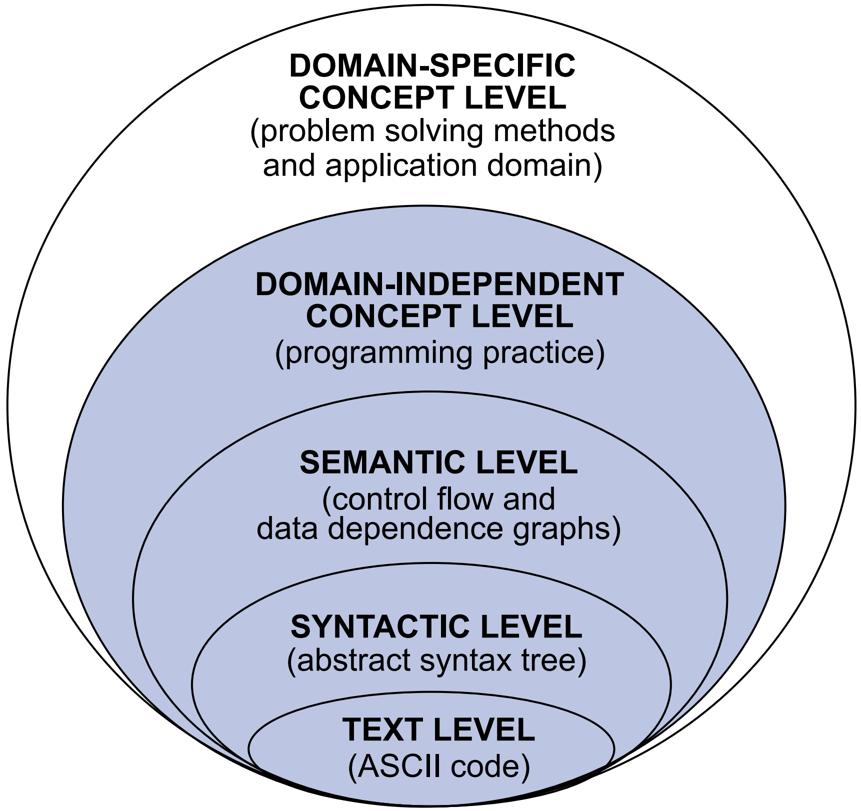
Constant propagation
Common subexpression elimination

Variable renaming

String matching & replacement

# The XARK Compiler Framework

- General and extensible solution for automatic kernel recognition at the domain-independent concept level.
- Properties:
  - Completeness: scalars/arrays/ pointers, ifs-endifs
  - Robustness: different versions of a kernel
  - Delocalization: statements spread over the source code
  - Uniqueness: one code, one kernel
  - Extensibility: user-defined kernels

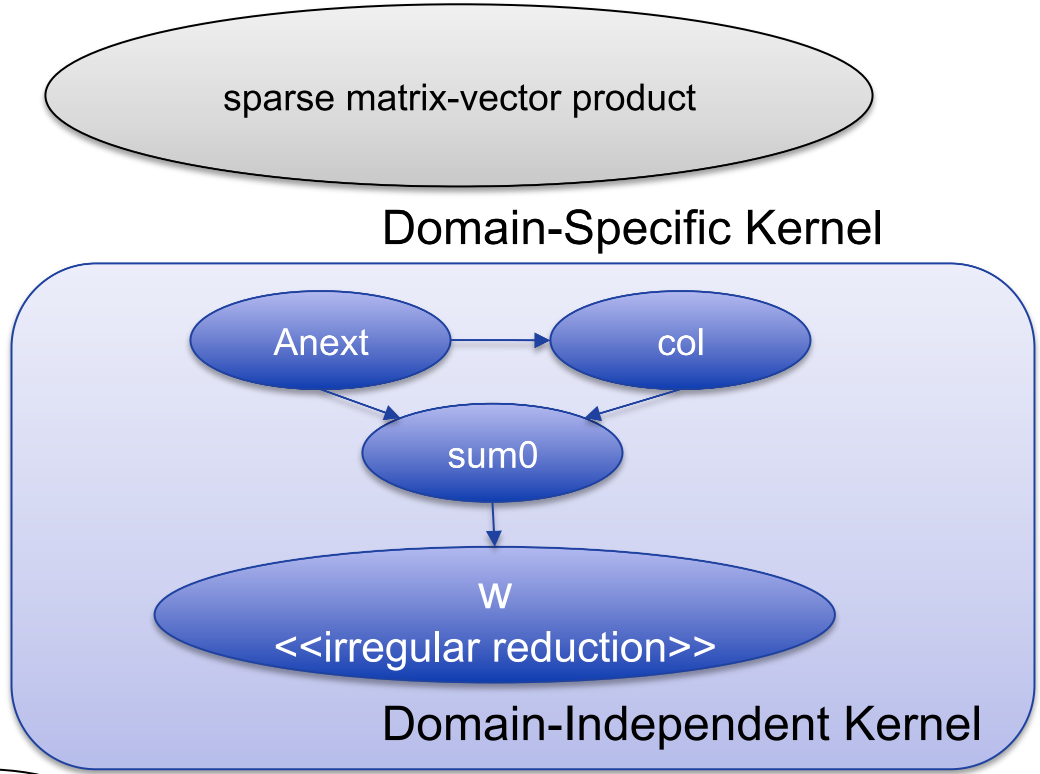



M. Arenaz, J. Touriño and R. Doallo: “XARK: An eXtensible framework for Automatic Recognition of computational Kernels“, ACM *Trans. Program. Lang. Syst.*, 30(6):1-56, October 2008

# Kernel Recognition

```
for(i) {
  Anext = …
  sum0 = A[Anext]…
  Anext++;
  while(Anext) {
    col = Acol[Anext];
    sum0 += A[Anext]…
    w[col][0] += …
    Anext++;
  }
  w[i][0] += sum0;
}
```

Excerpt of smvp() from EQUAKE





Forest of ASTs + DDG + CFG

# Index

- Introduction: Motivation & Foundations
- **Domain-Independent Kernel-based IR**
  - **Kernel-based Data Dependence Graph (KDDG)**
  - **Kernel-based Control Flow Graph (KCFG)**
- Automatic Partitioning
- Case Studies
  - Sobel Edge Filter
  - EQUAKE from SPEC CPU2000
- Conclusions & Future Work

# Kernel-based Data Dependence Graph (KDDG)

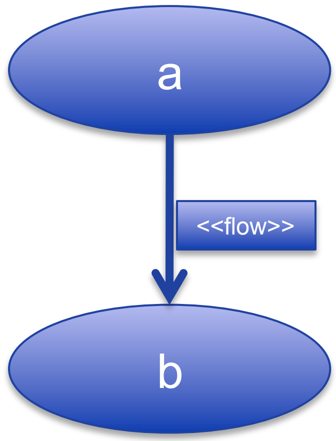
- KDDG = <N, E>
- N = set of kernels $K(x_1 \ldots x_n)$
  - $K(x_1 \ldots x_n)$ = sentences $s_1 \ldots s_n$ that define $x_1 \ldots x_n$ in Gated Single Assignment (GSA) form
    - Header of K: $s^h$ that dominates remaining statements
    - Latch of K: $s^l$ that posdominates remaining statements
- E = set of edges $K(x_1 \ldots x_n) \rightarrow K(y_1 \ldots y_m)$
  - Connect statements of different kernels

# Kernel-based Control Flow Graph (KCFG)

- $K(x_1 \ldots x_n) \rightarrow K(y_1 \ldots y_m)$ is a flow dependence if a dominance relationship exists
- $K_1$ dominates $K_2$ if and only if for all $s^2$ in $K_2$ there exists $s^1$ in $K_1$ such that
  1. $s^1$ and $s^2$ in same BB, then s1 precedes s2

  or

  2. $s^1$ in $BB_1$ and $s^2$ in $BB_2$, then $BB_1$ dominates $BB_2$ in Dominator Tree




```
a = 5;
b = a + 1;
```

```
a = 5;
if (c == 0) {
  b = a + 3;
} else {
  b = a + 1;
}
```

```
if (c == 0) {
  a = 5;
  b = a + 3;
} else {
  a = 2;
  b = a + 1;
}
```

```
if (c == 0) {
  a = 5;
  b = a + 3;
} else {
  b = 2;
}
```

# Construction of the KCFG

1. Group kernels into execution scopes

```
procedure COMPUTE_EXECUTION_SCOPES
    compute hierarchy of loops
    foreach kernel K in the KDDG do
        bb_dom = basic block of CFG that contains a stmt of K (excluding μ-stmt)
        foreach statement s^K in K do
            if s^K is not a μ-statement then
                bb_s^K = basic block of CFG that contains s^K
                if bb_s^K dominates bb_dom then
                    bb_dom = bb_s^K
                end if
            end if
        end for
        L = innermost enclosing loop of bb_dom
        if L includes loop indices that address the output variable of K then
            K.execution_scope = L
        end if
    end for
    remove non-attached loops from hierarchy
end procedure
```

# Construction of the KCFG

## 2. Search for flow dependences

**procedure** DETECT_FLOW_DEPENDENCES
    **foreach** kernel-level dependence $K_1 \rightarrow K_2$ of the KDDG **do**
        $L_1$ = execution_scope($K_1$); $L_2$ = execution_scope($K_2$)
        **if** ($L_1$.parent == $L_2$.parent) && ($L_1$ precedes $L_2$ in the hierarchy) **then**
            mark $K_1 \rightarrow K_2$ as flow dependence
        **else if** $K_1.latch$ dominates $K_2.header$ **then**
            mark $K_1 \rightarrow K_2$ as flow dependence
        **else**
            mark = true
            **foreach** $s^{K_2} \in K_2$ (excluding $\mu$-stmt) **do**
                dom_stmt_found = false
                **foreach** $s^{K_1} \in K_1$ (excluding $\mu$-stmt) **do**
                    $BB_1$ = basic_block($s^{K_1}$); $BB_2$ = basic_block($s^{K_2}$)
                    **if** ($BB_1 == BB_2$) && ($s^{K_1}$ precedes $s^{K_2}$) **then**
                        dom_stmt_found = true; break
                    **else if** ($BB_1 \neq BB_2$) && ($BB_1$ dominates $BB_2$) **then**
                        dom_stmt_found = true; break
                    **end if**
                **end for**
                mark = mark && dom_stmt_found
            **end for**
            **if** mark == true **then**
                mark $K_1 \rightarrow K_2$ as flow dependence
            **end if**
        **end if**
    **end for**
**end procedure**

# Index

- Introduction: Motivation & Foundations
- Domain-Independent Kernel-based IR
  - Kernel-based Data Dependence Graph (KDDG)
  - Kernel-based Control Flow Graph (KCFG)
- **Automatic Partitioning**
- Case Studies
  - Sobel Edge Filter
  - EQUAKE from SPEC CPU2000
- Conclusions & Future Work

# Automatic Partitioning

- Kernel-based IR exposes multiple levels of parallelism
  - intra- and inter-kernel
- Modern hardware architectures also expose multiple levels of parallelism
  - cluster, multicores, Intel SSE or AMD 3DNow!
- Kernel-based IR complexity is lower than statement-based IR
  - Exhaustive search (vs. heuristics)

# Automatic Partitioning

- Initialization
  - Kernels with low computational load ➔ non-splittable ➔ SIMD-like vector instructions
  - Merge consecutive execution scopes with one flow dependence
- Search best partitioning
  - Recursive function
  - Bottom-up traversal of KCFG looking for splittable kernels to be mapped to the system

# Automatic Partitioning

- Map kernels2arch
  - Create as many tasks as needed to fill-in a given number of processing elements with a given set of splittable kernels
  - Estimate cost
    - Computational load of the kernels
    - Computational capacity of processing elements
    - Amount of data that needs to be transferred
    - Synchronization
    - Etc.

# Index

- Introduction: Motivation & Foundations
- Domain-Independent Kernel-based IR
  - Kernel-based Data Dependence Graph (KDDG)
  - Kernel-based Control Flow Graph (KCFG)
- Automatic Partitioning
- **Case Studies**
  - **Sobel Edge Filter**
  - EQUAKE from SPEC CPU2000
- Conclusions & Future Work

# Case Study 1: Sobel

- The Sobel edge filter detects those pixels whose intensity is very different from the intensity of their neighbors
- Simple, but widely used in image processing and computer vision

# Sobel

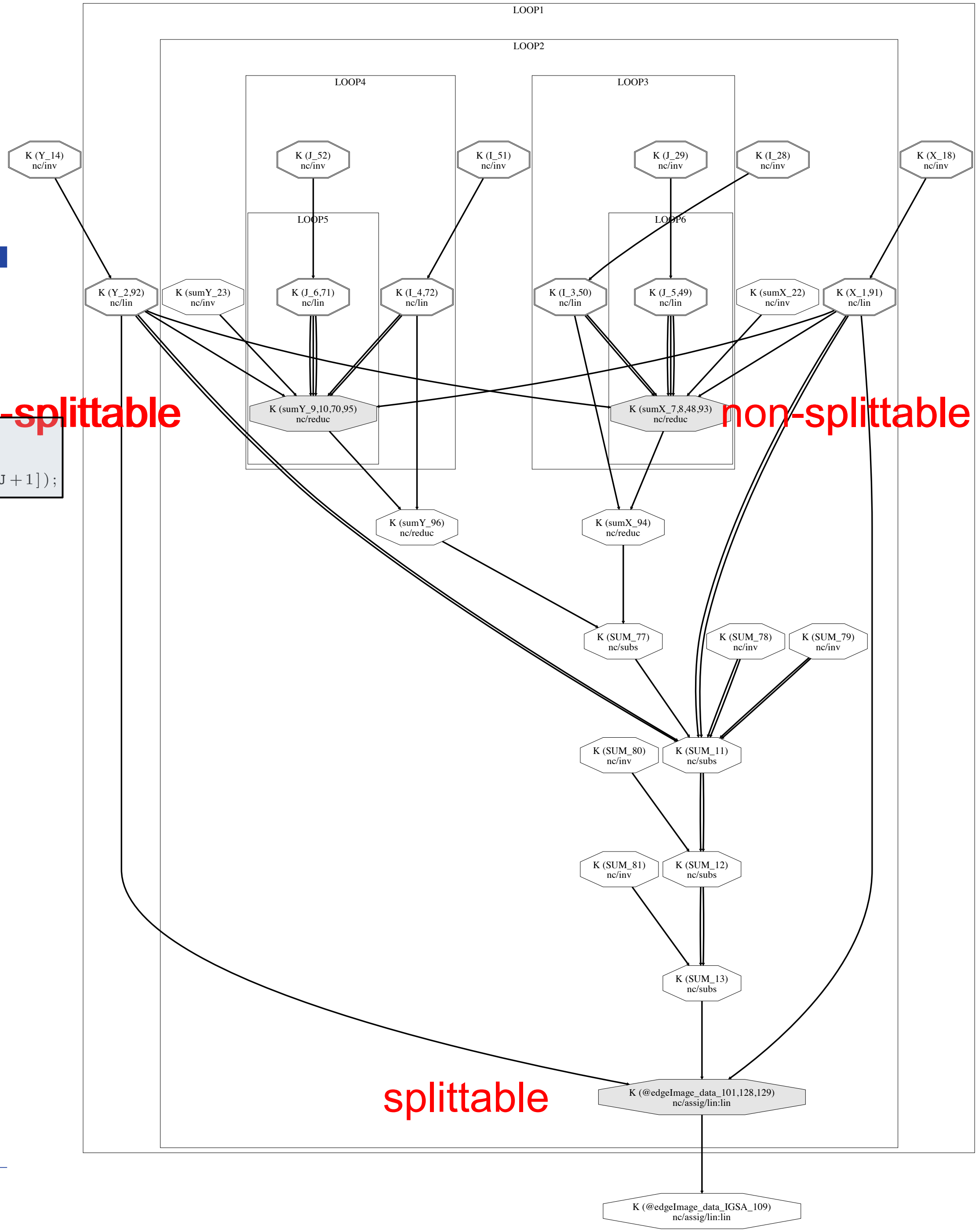
```c
void gradient_aprox (long *sum, unsigned char **data,
                     int cols, int Y, int X, int G[3][3])
{
  int I, J;
  for(I=-1; I<=1; I++)
    for(J=-1; J<=1; J++)
      (*sum) = (*sum) +
          (int)((*((*data) + X + I + (Y + J)*cols)) * G[I+1][J+1]);
}

int main(void)
{
  int             originalImage_rows, originalImage_cols,
  int             edgeImage_rows, edgeImage_cols;
  unsigned char*  originalImage_data, edgeImage_data;
  int             X, Y, I, J, GX[3][3], GY[3][3];
  long            sumX, sumY, SUM;

  for(Y=0; Y<=(originalImage_rows-1); Y++)  {
    for(X=0; X<=(originalImage_cols-1); X++)  {
      sumX = 0;
      sumY = 0;

      if(Y==0 || Y==originalImage_rows-1)
        SUM = 0;
      else if(X==0 || X==originalImage_cols-1)
        SUM = 0;
      else {
        gradient_aprox (&sumX, originalImage_data,
                        originalImage_cols, Y, X, GX);
        gradient_aprox (&sumY, originalImage_data,
                        originalImage_cols, Y, X, GY);
        SUM = abs(sumX) + abs(sumY);
      }
      if(SUM>255) SUM=255;
      if(SUM<0) SUM=0;

      *(edgeImage_data + X + Y*originalImage_cols) =
          255 - (unsigned char)(SUM);
    }
  }
}
```



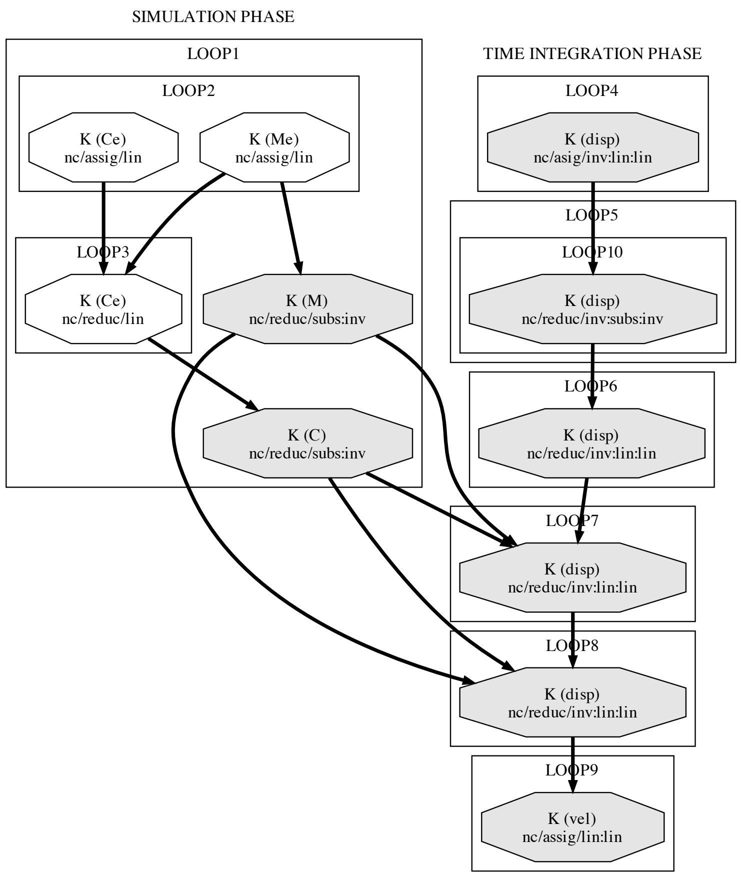
# Index

- Introduction: Motivation & Foundations
- Domain-Independent Kernel-based IR
  - Kernel-based Data Dependence Graph (KDDG)
  - Kernel-based Control Flow Graph (KCFG)
- Automatic Partitioning
- **Case Studies**
  - Sobel Edge Filter
  - **EQUAKE from SPEC CPU2000**
- Conclusions & Future Work

# Case Study 2: EQUAKE

- An example of full-scale application is EQUAKE, from SPEC CPU2000
- Simulation of seismic waves in large, highly heterogeneous valleys
- Finite element method
  - Simulation phase
  - Time integration phase
- 70 % of execution time is consumed by smvp()

# Case Study 2: EQUAKE




Initialization merges the execution scopes LOOP4-9

# Index

- Introduction: Motivation & Foundations
- Domain-Independent Kernel-based IR
  - Kernel-based Data Dependence Graph (KDDG)
  - Kernel-based Control Flow Graph (KCFG)
- Automatic Partitioning
- Case Studies
  - Sobel Edge Filter
  - EQUAKE from SPEC CPU2000
- **Conclusions & Future Work**

# Conclusions & Future Work

- Definition of a new kernel-based IR
  - Domain and codification-style independent
  - Inspired by standard statement-based IRs
  - Exposes multiple levels of parallelism
- Partitioning algorithm takes advantage of multiple levels in IR and architecture
  - Kernel-based IR allows exhaustive search
- Work in progress
  - Port of XARK from Polaris to GCC
    - From F77 to C, Fortran, and even C++, Java…
  - XARK is built on top of GSA form
    - Interprocedural GSA on top of GIMPLE-SSA
  - Implementation of automatic partitioning algorithm

# Conclusions & Future Work

- Future Work
  - Obtention of hardware characteristics
  - Estimation of the cost of a partition
  - Experimental evaluation with representative interprocedural implementations of well-known benchmarks

# Automatic Partitioning of Sequential Applications Driven by Domain-Independent Kernels

*José M. Andión*, Manuel Arenaz, and Juan Touriño

Computer Architecture Group
Department of Electronics and Systems
University of A Coruña
Spain