# FPGA-based Solution for Reverse Time Migration 中文介绍

## 1 简介

地质勘探的目的是为了挖掘地球内部的矿产资源，如石油，天然气。比如，如果要勘探海底的地址结构，轮船可以通过空气炮的形式向海底发射声波，同时在多个地方收集海底不同介质反射回来的地震波，对收集到的地震波进行处理，逆向生成海底的地质结构，从而判断是否有矿产资源的存在。过去使用的算法，比如 Ray Trace, 适用于较为简单的地质结构，如水层下面是盐层，盐层下面是岩石层，层的形状基本上为水平状，不存在大幅度的倾角；但面对较为复杂的地址结构，如盐层的倾角大于 70 度，层间复杂交错，传统的算法无法准确对此结构进行成像。

正如图 (1) 所示，单向的声波方程在边界较为陡峭的地方无法准确成像，这很有可能忽略了存在的矿产资源。偏移（Reverse Time Migration）算法能够对复杂的地质结构进行较为准确的成像，但是该算法的计算量非常大，CPU 版本的算法复杂度约为 O(n^3)，且伴随着非常大的常数，再加上需要处理的数据量非常大，通常是 TB 数量级，该算法在工业界的使用尚不广泛。但目前随着计算机计算力的提升以及加速卡，如 GPU，FPGA，的出现，为加速该算法提供了很大的可能性。本文利用 FPGA 作为加速器，在 FPGA 上实现该算法，目前的版本相对 8 核的 Intel i7 的 CPU 而言，得到了 6 倍的加速效果，且具有很大的优化空间。

Field Programming Gate Array (FPGA) 是一个可编程的硬件。通过对 FPGA 进行算法设计，代码直接通过硬件执行，而非通过操作系统的

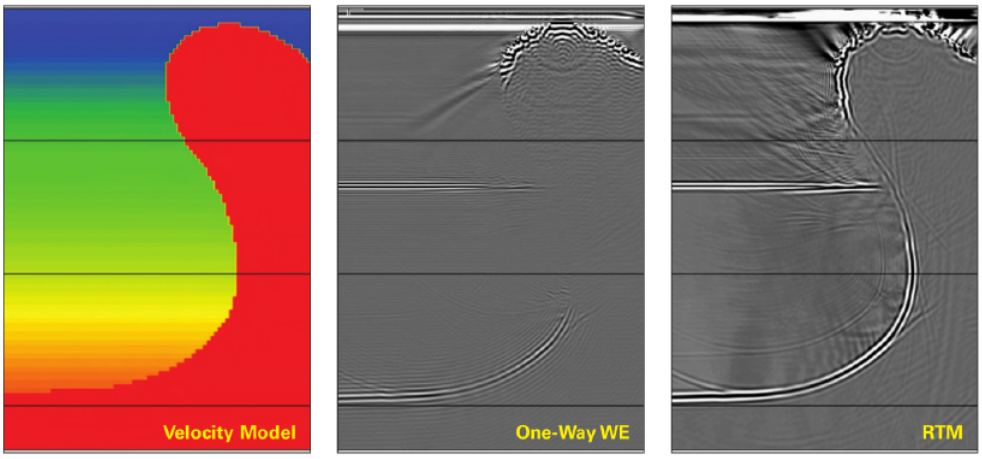


图 1: RTM 算法与单向便宜算法的比较

调度，因此具有非常高的效率。但在 FPGA 上设计算法是一件非常难的事情，传统的方法是通过硬件描述语言（HDL）对其进行编程，开发者需要关注每一个电路逻辑单元，这也是为什么 FPGA 较 CPU, GPU 或其他嵌入式设备而言，并没有那么流行的原因。本文利用 Xilinx 公司的 FPGA 芯片，Maxeler 公司的 MaxCompiler，在一个稍高的抽象层对 FPGA 进行设计，实现 RTM 算法。

# 2 研究内容

## 2.1 基于声波的正演 (Forward Modeling)

Reverse Time Migration 算法的核心是正演（forward modeling)。当地震波（声波）从波源发出后，从中心（波源）想四周扩散，当遇到边界（如盐层和岩石层之间的边界）时，同时进行反射和折射，以及部分能量被吸收。正演正是模拟这个过程。在计算机处理过程中，以一个网格划分的三维数组存储特定空间的波场 (wave field) ，正演的操作则是根据 $t$ 和 $t-1$ 时刻的波场，得出 $t+1$ 时刻的波场。设 $u$ 为当前时刻的波场，$c$ 为当前正演的速度模型（声波在特定区域的速度）, $s$ 为当前时刻波源的能

量，则正演使用的声波方程为

$$F = \frac{\partial^2 u}{\partial t^2} \quad = \quad c^2 \cdot (\bigtriangledown^2) u + s \tag{1}$$

其中，$\bigtriangledown$ 为 Laplace 算子，如果用二阶偏微分方程将其近似，可以得到如下的式子 (2)

$$\triangle u \quad = \quad \bigtriangledown^2 u = \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} + \frac{\partial^2 u}{\partial x^2} \tag{2}$$

将 Laplace 的近似代入 (1)，同时将 $\frac{\partial^2 u}{\partial t^2}$ 离散化，可以得到式子 (3)

$$\frac{u_{t+1} - 2u_t + u_{t-1}}{dt^2} \quad = \quad c_t^2 \cdot (\frac{\partial^2 u_t}{\partial x^2} + \frac{\partial^2 u_t}{\partial y^2} + \frac{\partial^2 u_t}{\partial x^2}) + s_t \tag{3}$$

在计算中，连续的偏微分方程可以通过有限差分的方法转换为差分方程，如此便可在计算网格中算出每个点的值。本文使用 stencil 方法对 $\frac{\partial^2 u_t}{\partial x^2} + \frac{\partial^2 u_t}{\partial y^2} + \frac{\partial^2 u_t}{\partial x^2}$ 在 $u(t, x, y, z)$ 进行求值。若采用 6 阶的 stencil，stencil 在点 $(x, y, z)$ 的值为

$$g(x, y, z) \quad = \quad \sum_{i=-3}^{+3} w_i f(i, y, z) + \sum_{i=-3}^{+3} w_i f(x, i, z) + \sum_{i=-3}^{+3} w_i f(x, y, i) - 2f(x, y, z)$$

图 (2) 勾画了 6 阶的三维 stencil 操作所要用到的点。

设研究的整个波场为 $x \times y \times z$ 的数组，则在每一次的迭代中，需要对 $x \cdot y \cdot z$ 个点进行 stencil 操作，在实际应用中，$x \approx y \approx z \approx 10^4$，迭代次数 $t \approx 10^4$，因此对于一次的波源冲击，需要进行 $10^{16}$ 浮点运算，这通常在 CPU 上是无法忍受的。况且现实生活中，往往有多次波源冲击以获得更加准确的数据。图 (3) 是基于声波方程的正演算法的伪代码，通过嵌套的循环数量可以推测该算法惊人的复杂度。

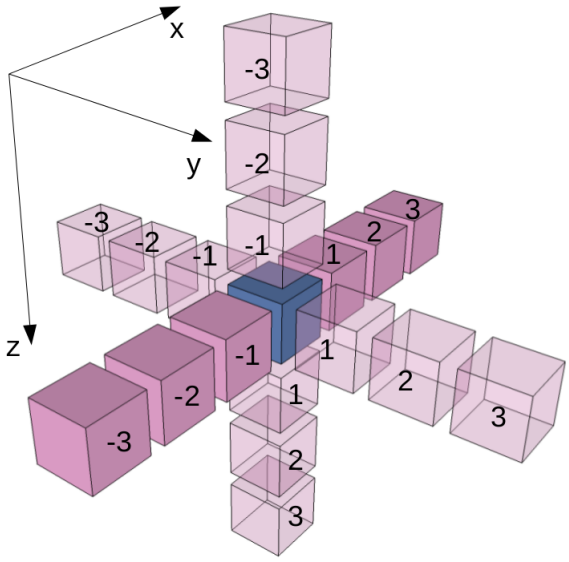


图 2: 6 阶三维 stencil 所涉及到的点

```
/* ns: # of shots
 * nt: # of time steps
 * nz: # of elements in z dimension
 * ny: # of elements in y dimension
 * nx: # of elements in x dimension
 * stencil: stencil operator for one element*/
for (is = 0; is < ns; i++)
  for (it = 1; it < nt - 1; it++)
    for (iz = 0; iz < nz; iz++)
      for (iy = 0; iy < ny; iy++)
        for (ix = 0; ix < nx; ix++) {
          u[it+1][iz][iy][ix] = 2*u[it][iz][iy][ix] - u[it-1][iz
              ][iy][ix] +
            dt*dt*(c*c*stencil(u[it][z][y][x]) + s[it]);
        }
```

图 3: 基于声波方程的正演算法的伪代码

## 2.2 基于声波的逆时偏移 (Revese Time Migration)

令从波源开始的正演方程为为 $F_{t=t_0}^{t_n}$，代表着从 $t_0$ 时刻开始正演，直到 $t_n$ 时刻结束。对每个接收点接收到的数据进行逆向正演 $F_{t=t_n}^{t_0}$, 代表着从 $t_n$ 时刻开始正演，逐步后退，直到 $t_0$ 时刻停止。而 Reverse Time Migration 则取每个点的正向正演和逆向正演的乘积作为改点的成像值。图 (4) 是基于声波方程的 RTM 算法的伪代码。对于网格中的每个点，需要进行两次正演，可见 RTM 算法的计算量约为单次正演计算量的两倍。

本文正是完成该算法在 FPGA 上的实现，以求获得比当前 CPU 实现更高的效率。

# 3 研究方法

本文的重心在于 RTM 算法在 FPGA 上的设计与实现，因此，省略了介绍如何搭建 FPGA 开发环境，如何调试及测试，尽管这是一个工作量大，学习曲线陡峭的过程。

## 3.1 FPGA 编程范式

FPGA 的编程范式是高级语言（C/C++) 开发者难以适应的。FPGA 展现在开发者（设计者）面前的是许多资源，如加法器，选择器，计数器，锁存器等等。开发者需要根据自己的需要，思考如何连接不通的器件，使其能够按照开发者的意图工作。这通常通过硬件描述语言（HDL）完成。个别厂商，如 Maxeler，为了方便一般程序员重新学习硬件描述语言，使用了 Java 语言对底层的元器件和功能进行封装并提供接口，因此，开发者可以通过购买 Maxeler 提供的开发工具，使用 Java 进行开发。但 Java 只是作为一种描述的语言，整个 FPGA 开发的范式与思想与传统的利用操作系统调度的程序截然不同。

FPGA 的设计不存在一般编程中的循环（while），分支（if/else）等控制指令，而是以流的形式完成计算的。这个因为 FPGA 中并不存在像 CPU 这样的处理器，能够完成复杂的跳转操作，只有许多简单的元器件，

```
/* ns: # of shots
 * nt: # of time steps
 * nz: # of elements in z dimension
 * ny: # of elements in y dimension
 * nx: # of elements in x dimension
 * s : source wave field array
 * r : receiver wave field array
 * res: the resulting image array, init as zeros
 * stencil: stencil operator for one element*/
for (is = 0; is < ns; i++) {
  /* forward modeling */
  for (it = 2; it < nt ; it++)
    for (iz = 0; iz < nz; iz++)
      for (iy = 0; iy < ny; iy++)
        for (ix = 0; ix < nx; ix++) {
          s(it,iz,iy,ix) = 2*s(it-1,iz,iy,ix) -
            s(it-2,iz,iy,ix) + dt*dt*c*c*stencil(s(it-1,z,y,x));

          if ((iz, iy, iz) corresponds to the source location)
            s(it,iz,iy,ix) += recored_source(it, iz, iy, ix);
          if ((iz, iy, ix) is in the boundary)
            process_boundary_condition();
        }

  /* reverse migration */
  for (it = nt - 1; it >= 2; it--)
    for (iz = 0; iz < nz; iz++)
      for (iy = 0; iy < ny; iy++)
        for (ix = 0; ix < nx; ix++) {
          r(it-2,iz,iy,ix) = 2*r(it-1,iz,iy,ix) -
            r(it,iz,iy,ix) + dt*dt*c*c*stencil(s(it-1,z,y,x));

          if ((iz, iy, iz) corresponds to the source location)
            r(it-2,iz,iy,ix) += recored_source(it-2,iz,iy,ix);
          if ((iz, iy, ix) is in the boundary)
            process_boundary_condition();

          // correlate the source and receiver wave field
          // into imaging results
          res(iz, iy, ix) += s(it-2,iz,iy,ix)*r(it-2,iz,iy,ix);
        }
}
```

图 4: 基于声波方程的 RTM 算法伪代码

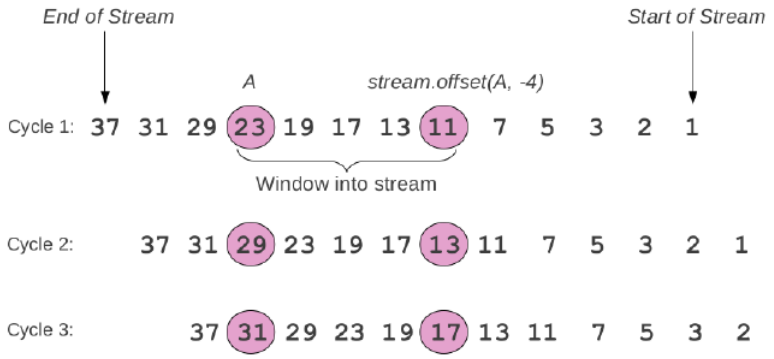


图 5: 3 个时钟周期的 stream offset 示例

元器件一旦连接完成，在每个时钟周期，每个输入既能产生对应的输出。因此，本文的难点也在于如何将图 (4) 伪代码中复杂的循环（for）和分支（if/else）以及数组元素访问指令通过简单的 FPGA 元器件完成。

## 3.2 数组元素访问

FPGA 中有许多元器件，也包括 DRAM，但是没有抽象意义上的数组，自然不存在所谓的数组元素访问。但绝大多数的问题都是需要通过数组完成的，如何在 FPGA 上完成数组的访问是迫切需要解决的问题。

由于 FPGA 的电路设计完成后，在每个时钟周期，一个输入对应一个输出，当前时钟周期能够访问的数组元素就是当前的输入。如果没有对当前的输入进行存储，下一个时钟周期的输入将覆盖当前的输入，因此当前的输入在以后的时钟周期中无法再次访问。针对 RTM 算法，如图 (3) 所示，每个时钟周期需要访问过去的三个数组元素（u[i-3], u[i-2], u[i-1]）和未来的三个元素（u[i+1], u[i+2], u[i+3]），这可以通过 FPGA 中的 stream offset 完成。

stream window 由锁存器组成，在当前时钟周期，stream window 储存着过去三个时钟周期的输入，同时取得未来三个时钟周期的输入。图 (5) 显示了三个时钟周期里，stream window 的变化情况。图 (5) 中，stream 中的第一个元素为 1，最后一个元素为 37，当前元素为 23，stream window

中储存着过去的 4 个元素，则可以通过 stream offset 的形式访问数组元素 u[i-4]。当进入下一个时钟周期时，stream window 以循环队列的形式，将队首的元素（11），弹出队列，同时压入最新的元素（29）。如此方式即可访问数组中的非自身元素。

该方法并不适应与所有数组元素的访问，适用于本算法是因为 RTM 算法的数组元素访问存在一定的规律性，该方法能够有效利用 FPGA 资源高效在一个时钟周期内完成元素的访问。

## 3.3 模拟循环（for）操作

如果面对的不是一维数组，而是二维或者三维数组，访问的元素就没有很强的规律性，如当前元素的前 3 个。RTM 算法中，每次需要判断该元素是否是边界元素，因此涉及到该元素的坐标 $(x, y, z)$。其中一种方法是将三维数组铺平，变成一维数组，但是计算点 $(x, y, z)$ 在一维数组中对应的坐标。显然，对于三维数组，这种计算极为复杂，例如，若 $(3, 3, 3) < (x, y, z) < (nx - 3, ny - 3, nz - 3)$ 代表该元素为非边界元素，将其展开围一维数组，则无法利用少量的判断条件来判断改点是不是边界点。因此，思考如何利用 FPGA 硬件来直接模拟循环是个不错的想法。

循环是高级语言编程中常见的控制操作，绝大多数的算法都需要循环来完成迭代和控制。但是 FPGA 中并不存在如此复杂的控制，如何通过元器件之间的组织和衔接来灵活模拟高级语言中的循环操作是本文实现的难点之一。

本文采用多个计数器（counter）共同协作的方式模拟循环操作。对于 RTM 算法的 3 维 stencil 操作，需要控制 x，y，z 三个维度的变量，则需要三个计数器来完成。例如，若需要完成如下嵌套循环操作

```
for (z = 0; z < nz; z++) {
  for (y = 0; y < ny; y++) {
    for (x = 0; x < nx; x++) {
      // operations here
    }
  }
}
```

则可以通过以下步骤完成。

1. 使用第一个计数器 counterX，用以模拟循环中 x 的行为。这个计数器的特征较为简单，计数器从 0 开始计数，每个时钟周期，计数器的值递增 1，当到达 nx-1 时，计数器重新返回 0（该操作成为 wrap）重新计数。该设计中，需要按照上述的特征描述计数器对应的参数。

2. 使用第二个计数器 counterY，该计数器具有与 counterX 类似的特征，唯一不同的是，counterY 并不是每个时钟周期递增 1，而是每经过 nx 个时钟周期递增 1。然而，FPGA 中的计数器总是会在下一个时钟周期到来时递增。因此采用额外的方式对其进行控制。对于每一个元器件，都可以设置其是否激活（enable），enable 状态的元器件会正常工作，而 disable 状态的元器件的不工作。在设计 counterY 时，将 counterX 与 counterY 级联（cascade），并将 counterX 的 wrap 输出脉冲连接到 counterY 的 enable 引脚。如此，则当 counterX 到达最大值，重新返回 0（wrap）时，wrap 输出的高电平电压能够是 counterY 处于 enable 状态，counterY 计数器递增 1。

3. 第三个计数器 counterZ 的设计方法与 counterY 的设计方式相同，将 counterY 与 counterZ 级联，并将 counterY 的 wrap 输出脉冲连接到 counterZ 的 enable 引脚。

上述的利用计数器的方法可以较为灵活的模拟循环操作。

### 3.4 模拟分支、逻辑与（&&）、逻辑或（||）操作

RTM 算法需要判断每个点是否为边界点，这需要复杂的分支和逻辑与判断操作，但是这些在 FPGA 中并不存在，只有通过元器件的方式模拟该操作。

简单的两路逻辑操作可以直接通过现成的 multiplexer 元器件完成，但面对复杂的判断操作，如

```
if (x >= 3 && x < nx - 3)
```

则无法通过一个 multiplexer 完成，因为每个 multiplexer 只有一个 select input，而 FPGA 中并不提供逻辑与（&&）和逻辑或（||）操作[1]，只有位与（&）和位或（|）。本文提出的方法是通过位与（&）和位或（|）来替代逻辑与（&&）和逻辑或（||）操作。

将 x >= 3 的输出与 x < nx-3 的输出作为位与（&）的输入，同时将位与（&）的输出作为 multiplexer 的 select input，则可以模拟上述的分支操作。

### 3.5 外部控制

由于并不是所有的 CPU 操作都能转换成 FPGA 操作，且根据 FPGA 的流处理特性，本文只将计算量最大的地方移到 FPGA 中进行计算。其他的控制，如控制 RTM 算法中的时间步，控制冲击波的次数，等外部循环将交由 CPU 控制。这当然存在优化的空间，可以是算法进一步得到加速。

## 4 研究结果与展望

通过将计算最密集的部分移植到 Xilinx FPGA 上，针对较小规模的数据进行实验验证和分析，然后运行时间与 8 核的 Intel i7 处理器相比，得到了 6 倍的性能提升。表 (1) 描述了两种实现的平台特性。与 CPU 相比，

[1]这两种操作在元器件的实现上必须存在短路，这在 FPGA 的流处理设计中是不现实的

| | Host (CPU) | Device (FPGA) |
|---|---|---|
| OS | Linux 2.6.18 | Maxeler OS 2011.3.1 |
| Compiler | GCC | MaxCompiler |
| Processor | Intel(R) Core(TM) i7 | Xilinx V6-SXT475 |
| Freq | 2.93GHz | 100MHz |
| RAM | DDR3 16G | DDR3 24G |

表 1: FPGA 与 CPU 的平台特征

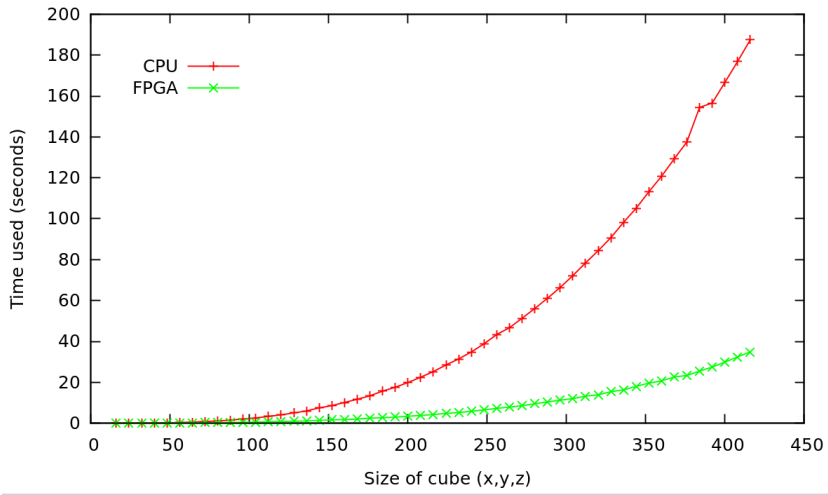


图 6: 比较当 size 变化时，CPU 和 FPGA 所花费的时间

FPGA 的主频非常低，这也决定了 FPGA 相对 CPU 而言具有非常低的功耗，这对于大规模的集群而言，具有深刻的意义，因为 CPU 的高主频将为公司带来巨额电费开销。

由于当前机器的局限性和为了考虑实验的效率，本文并未采用大规模的数据进行测试，当 time steps （迭代次数）为 10 时，波场传播空间的规模从 $16 \times 16 \times 16$ 到 $416 \times 416 \times 416$ 进行测试，将所耗费的时间（秒）画成曲线，得到如图 (6) 所示的结果。

从 (6) 可以看出，随着规模的增大，CPU 所花费的时间迅速增长，实际上，stencil 算法的复杂度为 $O(n^3)$，CPU 曲线也会以 $O(n^3)$ 的速度增

长，这在大数据量面前是难以忍受的。

相对 CPU 而言，FPGA 的实现方案优势较为明显，曲线增长的速率较 CPU 而言明显降低。目前尚未能准确分析 FPGA 实现的复杂度。

该 FPGA 实现并未经过任何优化，优化 FPGA 的实现需要对 FPGA 进行更加深入的认识，包括设计的技巧，资源的选择和利用等等。因此，本文的下一步工作则是针对 RTM 算法，在 FPGA 上逐步进行优化，以获得更高的效率。